P4$_{16}$ Language Specification

version 1.0.0

The P4 Language Consortium

2017-12-01

Abstract

P4 is a language for programming the data plane of network devices. This document provides a precise definition of the P4$_{16}$ language, which is the 2016 revision of the P4 language http://p4.org. The target audience for this document includes developers who want to write compilers, simulators, IDEs, and debuggers for P4 programs. This document may also be of interest to P4 programmers who are interested in understanding the syntax and semantics of the language at a deeper level.

Contents

1. Scope

This specification document defines the structure and interpretation of programs in the $P4_{16}$ language. It defines the syntax, semantic rules, and requirements for conformant implementations of the lan-

guage.

It does not define:

- Mechanisms by which P4 programs are compiled, loaded, and executed on packet-processing systems,
- Mechanisms by which data are received by one packet-processing system and delivered to another system,
- Mechanisms by which the control plane manages the match-action tables and other stateful objects defined by P4 programs,
- The size or complexity of P4 programs,
- The minimal requirements of packet-processing systems that are capable of providing a conformant implementation.

2. Terms, definitions, and symbols

Throughout this document, the following terms will be used:

- Architecture: A set of P4-programmable components and the data plane interfaces between them.
- Control plane: A class of algorithms and the corresponding input and output data that are concerned with the provisioning and configuration of the data plane.
- Data plane: A class of algorithms that describe transformations on packets by packet-processing systems.
- Metadata: Intermediate data generated during execution of a P4 program.
- Packet: A network packet is a formatted unit of data carried by a packet-switched network.
- Packet header: Formatted data at the beginning of a packet. A given packet may contain a sequence of packet headers representing different network protocols.
- Packet payload: Packet data that follows the packet headers.
- Packet-processing system: A data-processing system designed for processing network packets. In general, packet-processing systems implement control plane and data plane algorithms.
- Target: A packet-processing system capable of executing a P4 program.

All terms defined explicitly in this document should not be understood to refer implicitly to similar terms defined elsewhere. Conversely, any terms not defined explicitly in this document should be interpreted according to generally recognizable sources—e.g., IETF RFCs.

3. Overview

P4 is a language for expressing how packets are processed by the data plane of a programmable forwarding element such as a hardware or software switch, network interface card, router, or network appliance. The name P4 comes from the original paper that introduced the language, "Programming Protocol-independent Packet Processors," https://arxiv.org/pdf/1312.1719.pdf. While P4 was initially designed for programming switches, its scope has been broadened to cover a large variety of devices. In the rest of this document we use the generic term target for all such devices.

Many targets implement both a control plane and a data plane. P4 is designed to specify only the data plane functionality of the target. P4 programs also partially define the interface by which the control plane and the data-plane communicate, but P4 cannot be used to describe the control-plane func-

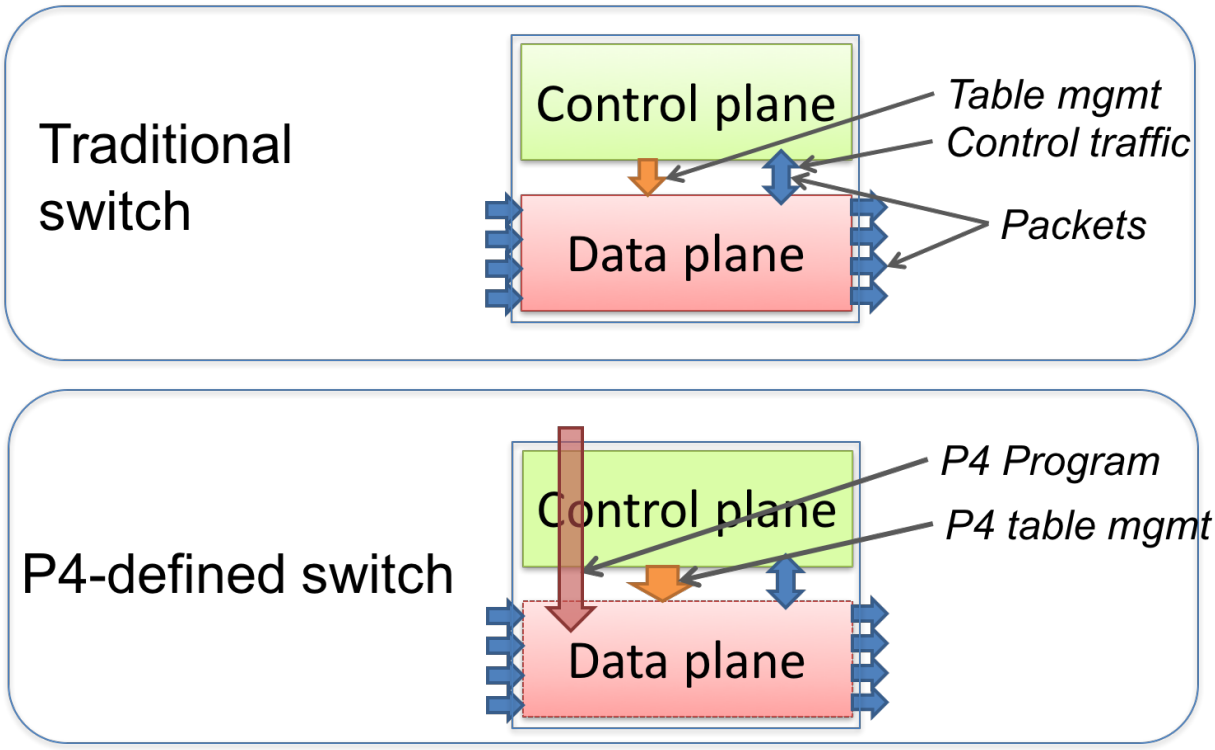

Figure 1. Traditional switches vs. programmable switches.

tionality of the target. In the rest of this document, when we talk about P4 as "programming a target", we mean "programming the data plane of a target".

As a concrete example of a target, Figure 1 illustrates the difference between a traditional fixed-function switch and a P4-programmable switch. In a traditional switch the manufacturer defines the data-plane functionality. The control-plane controls the data plane by managing entries in tables (e.g. routing tables), configuring specialized objects (e.g. meters), and by processing control-packets (e.g. routing protocol packets) or asynchronous events, such as link state changes or learning notifications.

A P4-programmable switch differs from a traditional switch in two essential ways:

- The data plane functionality is not fixed in advance but is defined by the a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program (shown by the long red arrow) and has no built-in knowledge of existing network protocols.
- The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

Hence, P4 can be said to be protocol independent, but it enables programmers to express a rich set of protocols and other data plane behaviors.

The core abstractions provided by the P4 language are:

- Header types describe the format (the set of fields and their sizes) of each header within a packet.
- Parsers describe the permitted sequences of headers within received packets, how to identify

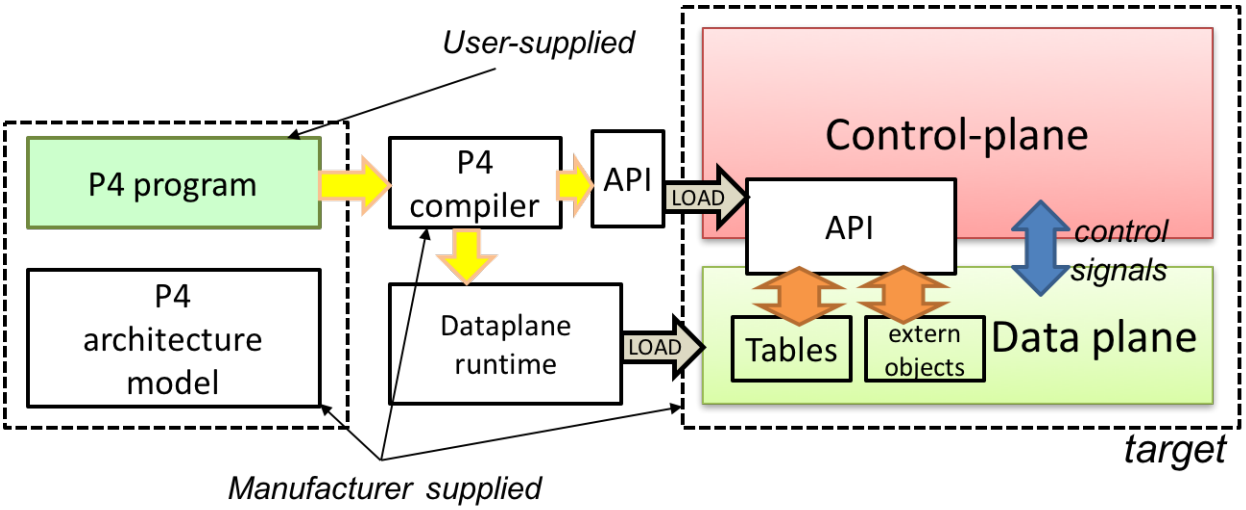

Figure 2. Programming a target with P4.

those header sequences, and the headers and fields to extract from packets.

- Tables associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multi-variable decisions.
- Actions are code fragments that describe how packet header fields and metadata are manipulated. Actions can include data, which is supplied by the control-plane at runtime.
- Match-action units perform the following sequence of operations:
  - Construct lookup keys from packet fields or computed metadata,
  - Perform table lookup using the constructed key, choosing an action (including the associated data) to execute, and
  - Finally, execute the selected action.
- Control flow expresses an imperative program that describes packet-processing on a target, including the data-dependent sequence of match-action unit invocations. Deparsing (packet reassembly) can also be performed using a control flow.
- Extern objects are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hard-wired (e.g., checksum units) and hence not programmable using P4.
- User-defined metadata: user-defined data structures associated with each packet.
- Intrinsic metadata: metadata provided by the architecture associated with each packet—e.g., the input port where a packet has been received.

Figure 2 shows a typical tool workflow when programming a target using P4.

Target manufacturers provide the hardware or software implementation framework, an architecture definition, and a P4 compiler for that target. P4 programmers write programs for a specific architecture, which defines a set of P4-programmable components on the target as well as their external data plane interfaces.

Compiling a set of P4 programs produces two artifacts:

- a data plane configuration that implements the forwarding logic described in the input program and
- an API for managing the state of the data plane objects from the control plane

P4 is a domain-specific language that is designed to be implementable on a large variety of targets including programmable network interface cards, FPGAs, software switches, and hardware ASICs. As such, the language is restricted to constructs that can be efficiently implemented on all of these platforms.

Assuming a fixed cost for table lookup operations and interactions with extern objects, all P4 programs (i.e., parsers and controls) execute a constant number of operations for each byte of an input packet received and analyzed. Although parsers may contain loops, provided some header is extracted on each cycle, the packet itself provides a bound on the total execution of the parser. In other words, under these assumptions, the computational complexity of a P4 program is linear in the total size of all headers, and never depends on the size of the state accumulated while processing data (e.g., the number of flows, or the total number of packets processed). These guarantees are necessary (but not sufficient) for enabling fast packet processing across a variety of targets.

P4 conformance of a target is defined as follows: if a specific target T supports only a subset of the P4 programming language, say P4^{T}, programs written in P4^{T} executed on the target should provide the exact same behavior as is described in this document. Note that P4 conformant targets can provide arbitrary P4 language extensions and `extern` elements.

3.1. Benefits of P4

Compared to state-of-the-art packet-processing systems (e.g., based on writing microcode on top of custom hardware), P4 provides a number of significant advantages:

- Flexibility: P4 makes many packet-forwarding policies expressible as programs, in contrast to traditional switches, which expose fixed-function forwarding engines to their users.
- Expressiveness: P4 can express sophisticated, hardware-independent packet processing algorithms using solely general-purpose operations and table look-ups. Such programs are portable across hardware targets that implement the same architectures (assuming sufficient resources are available).
- Resource mapping and management: P4 programs describe storage resources abstractly (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage low-level details such as allocation and scheduling.
- Software engineering: P4 programs provide important benefits such as type checking, information hiding, and software reuse.
- Component libraries: Component libraries supplied by manufacturers can be used to wrap hardware-specific functions into portable high-level P4 constructs.
- Decoupling hardware and software evolution: Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing.
- Debugging: Manufacturers can provide software models of an architecture to aid in the development and debugging of P4 programs.

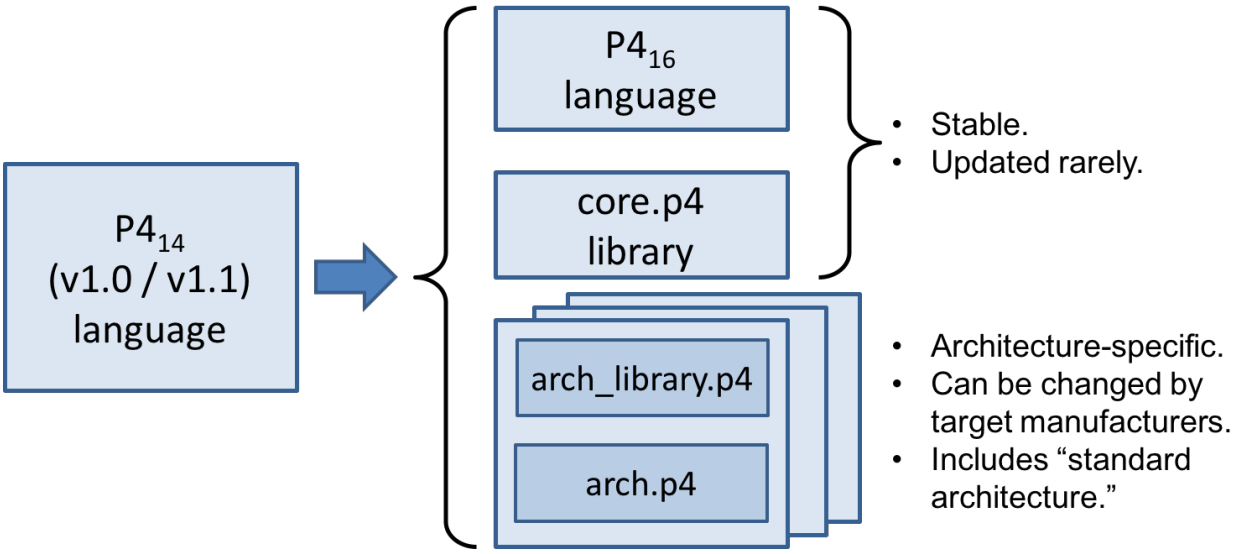

Figure 3. Evolution of the language between versions $P4_{14}$ (versions 1.0 and 1.1) and $P4_{16}$.

3.2. P4 language evolution: comparison to previous versions (P4 v1.0/v1.1)

Compared to $P4_{14}$, the earlier version of the language, $P4_{16}$ makes a number of significant, backwards-incompatible changes to the syntax and semantics of the language. The evolution from the previous version ($P4_{14}$) to the current one ($P4_{16}$) is depicted in Figure 3. In particular, a large number of language features have been eliminated from the language and moved into libraries including counters, checksum units, meters, etc.

Hence, the language has been transformed from a complex language (more than 70 keywords) into a relatively small core language (less than 40 keywords, shown in Section A) accompanied by a library of fundamental constructs that are needed for writing most P4.

The v1.1 version of P4 introduced a language construct called `extern` that can be used to describe library elements. Many constructs defined in the v1.1 language specification will thus be transformed into such library elements (including constructs that have been eliminated from the language, such as counters and meters). Some of these `extern` objects are expected to be standardized, and they will be in the scope of a future document describing a standard library of P4 elements. In this document we provide several examples of `extern` constructs. $P4_{16}$ also introduces and repurposes some v1.1 language constructs for describing the programmable parts of an architecture. These language constructs are: `parser`, `state`, `control`, and `package`.

One important goal of the $P4_{16}$ language revision is to provide a stable language definition. In other words, we strive to ensure that all programs written in $P4_{16}$ will remain syntactically correct and behave identically when treated as programs for future versions of the language. Moreover, if some future version of the language requires breaking backwards compatibility, we will seek to provide an easy path for migrating $P4_{16}$ programs to the new version.

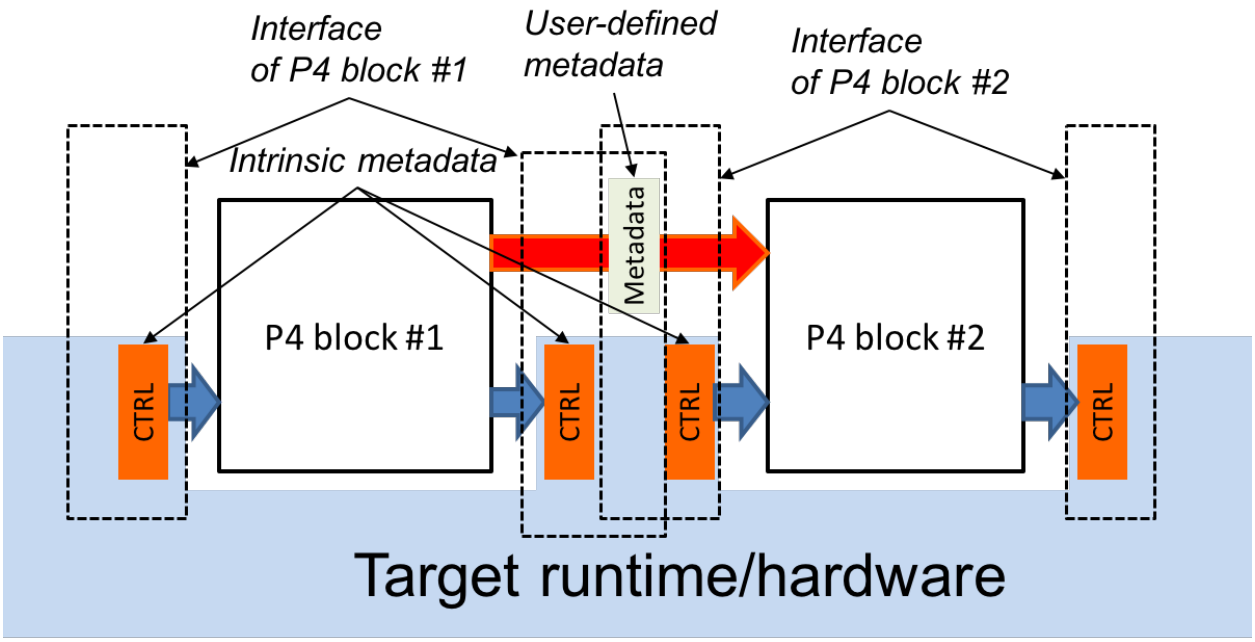

Figure 4. P4 program interfaces.

4. Architecture Model

The P4 architecture identifies the P4-programmable blocks (e.g., parser, ingress control flow, egress control flow, deparser, etc.) and their data plane interfaces.

The P4 architecture can be thought of as a contract between the program and the target. Each manufacturer must therefore provide both a P4 compiler as well as an accompanying architecture definition for their target. (We expect that P4 compilers can share a common front-end that handles all architectures). The architecture definition does not have to expose the entire programmable surface of the data plane—a manufacturer may even choose to provide multiple definitions for the same hardware device, each with different capabilities (e.g., with or without multicast support).

Figure 4 illustrates the data plane interfaces between P4-programmable blocks. It shows a target that has two programmable blocks (#1 and #2). Each block is programmed through a separate fragment of P4 code. The target interfaces with the P4 program through a set of control registers or signals. Input controls provide information to P4 programs (e.g., the input port that a packet was received from), while output controls can be written to by P4 programs to influence the target behavior (e.g., the output port where a packet has to be directed). Control registers/signals are represented in P4 as intrinsic metadata. P4 programs can also store and manipulate data pertaining to each packet as user-defined metadata.

The behavior of a P4 program can be fully described in terms of transformations that map vectors of bits to vectors of bits. To actually process a packet, the architecture model interprets the bits that the P4 program writes to intrinsic metadata. For example, to cause a packet to be forwarded on a specific output port, a P4 program may need to write the index of an output port into a dedicated control register. Similarly, to cause a packet to be dropped, a P4 program may need to set a “drop” bit into another dedicated control register. Note that the details of how intrinsic metadata are interpreted is

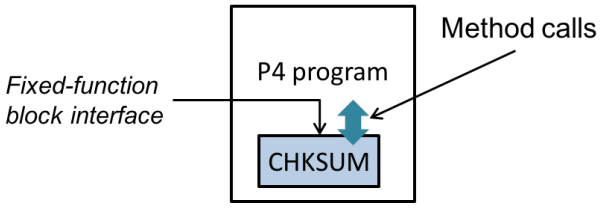

Figure 5. P4 program invoking the services of a fixed-function object.

architecture-specific.

P4 programs can invoke services implemented by extern objects and functions provided by the architecture. Figure 5 depicts a P4 program invoking the services of a built-in checksum computation unit on a target. The implementation of the checksum unit is not specified in P4, but its interface is. In general, the interface for an extern object describes each operation it provides, as well as their parameter and return types.

In general, P4 programs are not expected to be portable across different architectures. For example, executing a P4 program that broadcasts packets by writing into a custom control register will not function correctly on a target that does not have the control register. However, P4 programs written for a given architecture should be portable across all targets that faithfully implement the corresponding model, provided there are sufficient resources.

4.1. Standard architectures

We expect that the P4 community will evolve a small set of standard architecture models pertaining to specific verticals. Wide adoption of such standard architectures will promote portability of P4 programs across different targets. However, defining these standard architectures is outside of the scope of this document.

4.2. Data plane interfaces

To describe a functional block that can be programmed in P4, the architecture includes a type declaration that specifies the interfaces between the block and the other components in the architecture. For example, the architecture might contain a declaration such as the following:

```
control MatchActionPipe<H>(in bit<4> inputPort,
                           inout H parsedHeaders,
                           out bit<4> outputPort);
```

This type declaration describes a block named `MatchActionPipe` that can be programmed using a data-dependent sequence of match-action unit invocations and other imperative constructs (indicated by the `control` keyword). The interface between the `MatchActionPipe` block and the other components of the architecture can be read off from this declaration:

- The first parameter is a 4-bit value named `inputPort`. The direction `in` indicates that this parameter is an input that cannot be modified.
- The second parameter is an object of type `H` named `parsedHeaders`, where `H` is a type variable representing the headers that will be defined later by the P4 programmer. The direction `inout` indicates

that this parameter is both an input and an output.
- The third parameter is a 4-bit value named `outputPort`. The direction `out` indicates that this parameter is an output whose value is undefined initially but can be modified.

4.3. Extern objects and functions

P4 programs can also interact with objects and functions provided by the architecture. Such objects are described using the `extern` construct, which describes the interfaces that such objects expose to the data-plane.

An `extern` object describes a set of methods that are implemented by an object, but not the implementation of these methods (i.e., it is similar to an abstract class in an object-oriented language). For example, the following construct could be used to describe the operations offered by an incremental checksum unit:

```
extern Checksum16 {
     Checksum16();                // constructor
     void clear();                // prepare unit for computation
     void update<T>(in T data); // add data to checksum
     void remove<T>(in T data); // remove data from existing checksum
     bit<16> get(); // get the checksum for the data added since last clear
}
```

5. Example: A very simple switch

As an example to illustrate the features of architectures, consider implementing a very simple switch in P4. We will first describe the architecture of the switch and then write a complete P4 program that specifies the data plane behavior of the switch. This example demonstrates many important features of the P4 programming language.

We call our architecture the "Very Simple Switch" (VSS). Figure 6 is a diagram of this architecture. There is nothing inherently special about VSS—it is just a pedagogical example that illustrates how programmable switches can be described and programmed in P4. VSS has a number of fixed-function blocks (shown in cyan in our example), whose behavior is described in Section 5.2. The white blocks are programmable using P4.

VSS receives packets through one of 8 input Ethernet ports, through a recirculation channel, or from a port connected directly to the CPU. VSS has one single parser, feeding into a single match-action pipeline, which feeds into a single deparser. After exiting the deparser, packets are emitted through one of 8 output Ethernet ports or one of 3 "special" ports:

- Packets sent to the "CPU port" are sent to the control plane
- Packets sent to the "Drop port" are discarded
- Packets sent to the "Recirculate port" are re-injected in the switch through a special input port

The white blocks in the figure are programmable, and the user must provide a corresponding P4 program to specify the behavior of each such block. The red arrows indicate the flow of user-defined data. The cyan blocks are fixed-function components. The green arrows are data plane interfaces used to convey information between the fixed-function blocks and the programmable blocks—exposed in the P4 program as intrinsic metadata.

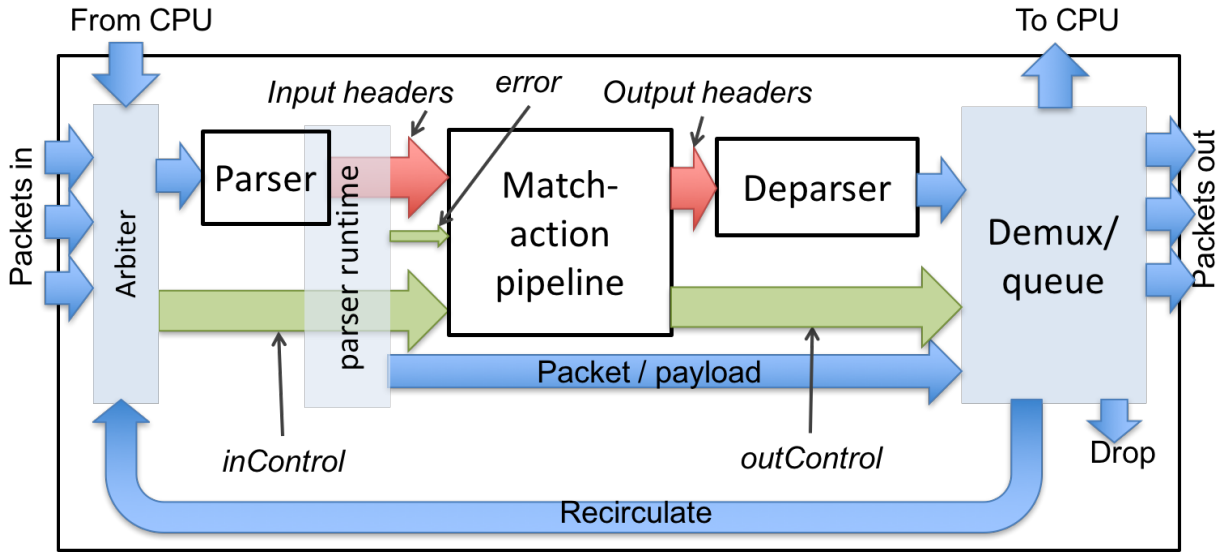

Figure 6. The Very Simple Switch (VSS) architecture.

5.1. Very Simple Switch Architecture

The following P4 program provides a declaration of VSS in P4, as it would be provided by the VSS manufacturer. The declaration contains several type declarations, constants, and finally declarations for the three programmable blocks; the code uses syntax highlighting. The programmable blocks are described by their types; the implementation of these blocks has to be provided by the switch programmer.

```
// File "very_simple_switch_model.p4"
// Very Simple Switch P4 declaration
// core library needed for packet_in and packet_out definitions
# include <core.p4>
/* Various constants and structure declarations */
/* ports are represented using 4-bit values */
typedef bit<4> PortId;
/* only 8 ports are "real" */
const PortId REAL_PORT_COUNT = 4w8;  // 4w8 is the number 8 in 4 bits
/* metadata accompanying an input packet */
struct InControl {
    PortId inputPort;
}
/* special input port values */
const PortId RECIRCULATE_IN_PORT = 0xD;
const PortId CPU_IN_PORT = 0xE;
/* metadata that must be computed for outgoing packets */
struct OutControl {
    PortId outputPort;
```

```
}
/* special output port values for outgoing packet */
const PortId DROP_PORT = 0xF;
const PortId CPU_OUT_PORT = 0xE;
const PortId RECIRCULATE_OUT_PORT = 0xD;
/* Prototypes for all programmable blocks */
/**
 * Programmable parser.
 * @param <H> type of headers; defined by user
 * @param b input packet
 * @param parsedHeaders headers constructed by parser
 */
parser Parser<H>(packet_in b,
                 out H parsedHeaders);
/**
 * Match-action pipeline
 * @param <H> type of input and output headers
 * @param headers headers received from the parser and sent to the deparser
 * @param parseError error that may have surfaced during parsing
 * @param inCtrl information from architecture, accompanying input packet
 * @param outCtrl information for architecture, accompanying output packet
 */
control Pipe<H>(inout H headers,
                in error parseError,// parser error
                in InControl inCtrl,// input port
                out OutControl outCtrl); // output port
/**
 * VSS deparser.
 * @param <H> type of headers; defined by user
 * @param b output packet
 * @param outputHeaders headers for output packet
 */
control Deparser<H>(inout H outputHeaders,
                    packet_out b);
/**
 * Top-level package declaration - must be instantiated by user.
 * The arguments to the package indicate blocks that
 * must be instantiated by the user.
 * @param <H> user-defined type of the headers processed.
 */
package VSS<H>(Parser<H> p,
               Pipe<H> map,
               Deparser<H> d);
// Architecture-specific objects that can be instantiated
// Checksum unit
extern Checksum16 {
```

```
    Checksum16();                // constructor
    void clear();                // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

Let us describe some of these elements:

- The included file `core.p4` is described in more detail in Appendix B. It defines some standard data-types and error codes.
- `bit<4>` is the type of bit-strings with 4 bits.
- The syntax `4w0xF` indicates the value 15 represented using 4 bits. An alternative notation is `4w15`. In many circumstances the width modifier can be omitted, writing just `15`.
- `error` is a built-in P4 type for holding error codes
- Next follows the declaration of a parser:

  ```
  parser Parser<H>(packet_in b, out H parsedHeaders);
  ```

  This declaration describes the interface for a parser, but not yet its implementation, which will be provided by the programmer. The parser reads its input from a `packet_in`, which is a pre-defined P4 extern object that represents an incoming packet, declared in the `core.p4` library. The parser writes its output (the `out` keyword) into the `parsedHeaders` argument. The type of this argument is `H`, yet unknown—it will also be provided by the programmer.
- The declaration

  ```
  control Pipe<H>(inout H headers,
                  in error parseError,
                  in InControl inCtrl,
                  out OutControl outCtrl);
  ```

  describes the interface of a Match-Action pipeline named `Pipe`.

The pipeline receives three inputs: the headers `headers`, a parser error `parseError`, and the `inCtrl` control data. Figure 6 indicates the different sources of these pieces of information. The pipeline writes its outputs into `outCtrl`, and it must update in place the headers to be consumed by the deparser.

- The top-level package is called `VSS`; in order to program a VSS, the user will have to instantiate a package of this type (shown in the next section). The top-level package declaration also depends on a type variable H:

```
package VSS<H>
```

A type variable indicates a type yet unknown that must be provided by the user at a later time. In this case `H` is the type of the set of headers that the user program will be processing; the parser will produce the parsed representation of these headers, and the match-action pipeline will update the input headers in place to produce the output headers.

- The `package VSS` declaration has three complex parameters, of types `Parser`, `Pipe`, and `Deparser` respectively; which are exactly the declarations we have just described. In order to program the target one has to supply values for these parameters.
- In this program the `inCtrl` and `outCtrl` structures represent control registers. The content of the headers structure is stored in general-purpose registers.
- The `extern Checksum16` declaration describes an extern object whose services can be invoked to compute checksums.

5.2. Very Simple Switch Architecture Description

In order to fully understand VSS's behavior and write meaningful P4 programs for it, and for implementing a control plane, we also need a full behavioral description of the fixed-function blocks. This section can be seen as a simple example illustrating all the details that have to be handled when writing an architecture description. The P4 language is not intended to cover the description of all such functional blocks—the language can only describe the interfaces between programmable blocks and the architecture. For the current program, this interface is given by the `Parser`, `Pipe`, and `Deparser` declarations. In practice we expect that the complete description of the architecture will be provided as an executable program and/or diagrams and text; in this document we will provide informal descriptions in English.

5.2.1. Arbiter block

The input arbiter block performs the following functions:

- It receives packets from one of the physical input Ethernet ports, from the control plane, or from the input recirculation port.
- For packets received from Ethernet ports, the block computes the Ethernet trailer checksum and verifies it. If the checksum does not match, the packet is discarded. If the checksum does match, it is removed from the packet payload.
- Receiving a packet involves running an arbitration algorithm if multiple packets are available.
- If the arbiter block is busy processing a previous packet and no queue space is available, input ports may drop arriving packets, without indicating the fact that the packets were dropped in any way.
- After receiving a packet, the arbiter block sets the `inCtrl.inputPort` value that is an input to the match-action pipeline with the identity of the input port where the packet originated. Physical Ethernet ports are numbered 0 to 7, while the input recirculation port has a number 13 and the CPU port has the number 14.

5.2.2. Parser runtime block

The parser runtime block works in concert with the parser. It provides an error code to the match-action pipeline, based on the parser actions, and it provides information about the packet payload (e.g., the size of the remaining payload data) to the demux block. As soon as a packet's processing is completed by the parser, the match-action pipeline is invoked with the associated metadata as inputs (packet headers and user-defined metadata).

5.2.3. Demux block

The core functionality of the demux block is to receive the headers for the outgoing packet from the deparser and the packet payload from the parser, to assemble them into a new packet and to send the result to the correct output port. The output port is specified by the value of `outCtrl.ouputPort`, which is set by the match-action pipeline.

- Sending the packet to the drop port causes the packet to disappear.
- Sending the packet to an output Ethernet port numbered between 0 and 7 causes it to be emitted on the corresponding physical interface. The packet may be placed in a queue if the output interface is already busy emitting another packet. When the packet is emitted, the physical interface computes a correct Ethernet checksum trailer and appends it to the packet.
- Sending a packet to the output CPU port causes the packet to be transferred to the control plane. In this case, the packet that is sent to the CPU is the original input packet, and not the packet received from the deparser—the latter packet is discarded.
- Sending the packet to the output recirculation port causes it to appear at the input recirculation port. Recirculation is useful when packet processing cannot be completed in a single pass.
- If the `outputPort` has an illegal value (e.g., 9), the packet is dropped.
- Finally, if the demux unit is busy processing a previous packet and there is no capacity to queue the packet coming from the deparser, the demux unit may drop the packet, irrespective of the output port indicated.

Please note that some of the behaviors of the demux block may be unexpected—we have highlighted them in bold. We are not specifying here several important behaviors related to queue size, arbitration, and timing, which also influence the packet processing.

The arrow shown from the parser runtime to the demux block represents an additional information flow from the parser to the demux: the packet being processed as well as the offset within the packet where parsing ended (i.e., the start of the packet payload).

5.2.4. Available extern blocks

The VSS architecture provides an incremental checksum extern block, called `Checksum16`. The checksum unit has a constructor and four methods:

- `clear()`: prepares the unit for a new computation
- `update<T>(in T data)`: add some data to be checksummed. The data must be either a bit-string, a header-typed value, or a `struct` containing such values. The fields in the header/struct are concatenated in the order they appear in the type declaration.
- `get()`: returns the 16-bit one's complement checksum. When this function is invoked the checksum must have received an integral number of bytes of data.

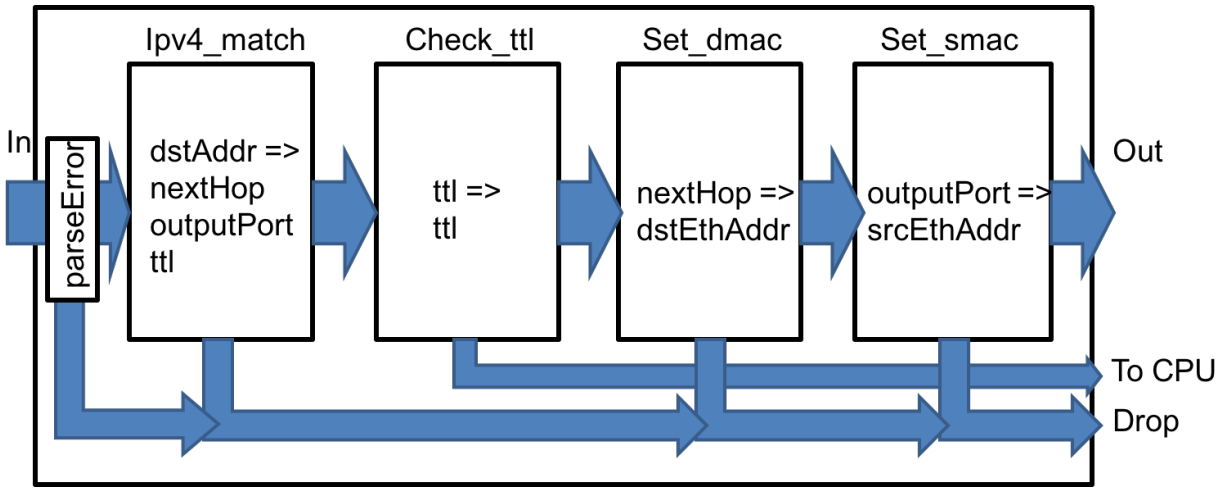

Figure 7. Diagram of the match-action pipeline expressed by the VSS P4 program.

- `remove<T>(in T data)`: assuming that `data` was used for computing the current checksum, `data` is removed from the checksum.

5.3. A complete Very Simple Switch program

Here we provide a complete P4 program that implements basic forwarding for IPv4 packets on the VSS architecture. This program does not utilize all of the features provided by the architecture—e.g., recirculation—but it does use preprocessor `#include` directives (see Section 6.2).

The parser attempts to recognize an Ethernet header followed by an IPv4 header. If either of these headers are missing, parsing terminates with an error. Otherwise it extracts the information from these headers into a `Parsed_packet` structure. The match-action pipeline is shown in Figure 7; it comprises four match-action units (represented by the P4 `table` keyword):

- If any parser error has occurred, the packet is dropped (i.e., by assigning `outputPort` to `DROP_PORT`)
- The first table uses the IPv4 destination address to determine the `outputPort` and the IPv4 address of the next hop. If this lookup fails, the packet is dropped. The table also decrements the IPv4 `ttl` value.
- The second table checks the `ttl` value: if the `ttl` becomes 0, the packet is sent to the control plane through the CPU port.
- The third table uses the IPv4 address of the next hop (which was computed by the first table) to determine the Ethernet address of the next hop.
- Finally, the last table uses the `outputPort` to identify the source Ethernet address of the current switch, which is set in the outgoing packet.

The deparser constructs the outgoing packet by reassembling the Ethernet and IPv4 headers as computed by the pipeline.

```
// Include P4 core library
# include <core.p4>
```

```
// Include very simple switch architecture declarations
# include "very_simple_switch_model.p4"

// This program processes packets comprising an Ethernet and an IPv4
// header, and it forwards packets using the destination IP address

typedef bit<48>  EthernetAddress;
typedef bit<32>  IPv4Address;

// Standard Ethernet header
header Ethernet_h {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16>          etherType;
}

// IPv4 header (without options)
header IPv4_h {
    bit<4>        version;
    bit<4>        ihl;
    bit<8>        diffserv;
    bit<16>       totalLen;
    bit<16>       identification;
    bit<3>        flags;
    bit<13>       fragOffset;
    bit<8>        ttl;
    bit<8>        protocol;
    bit<16>       hdrChecksum;
    IPv4Address  srcAddr;
    IPv4Address  dstAddr;
}

// Structure of parsed headers
struct Parsed_packet {
    Ethernet_h ethernet;
    IPv4_h     ip;
}

// Parser section

// User-defined errors that may be signaled during parsing
error {
    IPv4OptionsNotSupported,
    IPv4IncorrectVersion,
    IPv4ChecksumError
}
```

```
parser TopParser(packet_in b, out Parsed_packet p) {
    Checksum16() ck;  // instantiate checksum unit

    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x0800: parse_ipv4;
            // no default rule: all other packets rejected
        }
    }

    state parse_ipv4 {
        b.extract(p.ip);
        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
        verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
        ck.clear();
        ck.update(p.ip);
        // Verify that packet checksum is zero
        verify(ck.get() == 16w0, error.IPv4ChecksumError);
        transition accept;
    }
}

// Match-action pipeline section

control TopPipe(inout Parsed_packet headers,
                in error parseError, // parser error
                in InControl inCtrl, // input port
                out OutControl outCtrl) {
     IPv4Address nextHop;  // local variable

     /**
      * Indicates that a packet is dropped by setting the
      * output port to the DROP_PORT
      */
      action Drop_action() {
          outCtrl.outputPort = DROP_PORT;
      }

     /**
      * Set the next hop and the output port.
      * Decrements ipv4 ttl field.
      * @param ivp4_dest ipv4 address of next hop
      * @param port output port
      */
```

```
    action Set_nhop(IPv4Address ipv4_dest, PortId port) {
        nextHop = ipv4_dest;
        headers.ip.ttl = headers.ip.ttl - 1;
        outCtrl.outputPort = port;
    }

    /**
     * Computes address of next IPv4 hop and output port
     * based on the IPv4 destination of the current packet.
     * Decrements packet IPv4 TTL.
     * @param nextHop IPv4 address of next hop
     */
    table ipv4_match {
        key = { headers.ip.dstAddr: lpm; }  // longest-prefix match
        actions = {
            Drop_action;
            Set_nhop;
        }
        size = 1024;
        default_action = Drop_action;
    }

    /**
     * Send the packet to the CPU port
     */
    action Send_to_cpu() {
        outCtrl.outputPort = CPU_OUT_PORT;
    }

    /**
     * Check packet TTL and send to CPU if expired.
     */
    table check_ttl {
        key = { headers.ip.ttl: exact; }
        actions = { Send_to_cpu; NoAction; }
        const default_action = NoAction; // defined in core.p4
    }

    /**
     * Set the destination MAC address of the packet
     * @param dmac destination MAC address.
     */
    action Set_dmac(EthernetAddress dmac) {
        headers.ethernet.dstAddr = dmac;
    }
```

```
    /**
     * Set the destination Ethernet address of the packet
     * based on the next hop IP address.
     * @param nextHop IPv4 address of next hop.
     */
     table dmac {
          key = { nextHop: exact; }
          actions = {
                Drop_action;
                Set_dmac;
          }
          size = 1024;
          default_action = Drop_action;
     }

     /**
      * Set the source MAC address.
      * @param smac: source MAC address to use
      */
      action Set_smac(EthernetAddress smac) {
           headers.ethernet.srcAddr = smac;
      }

     /**
      * Set the source mac address based on the output port.
      */
     table smac {
           key = { outCtrl.outputPort: exact; }
           actions = {
                 Drop_action;
                 Set_smac;
          }
          size = 16;
          default_action = Drop_action;
     }

     apply {
          if (parseError != error.NoError) {
               Drop_action();  // invoke drop directly
               return;
          }

          ipv4_match.apply(); // Match result will go into nextHop
          if (outCtrl.outputPort == DROP_PORT) return;

          check_ttl.apply();
```

```
            if (outCtrl.outputPort == CPU_OUT_PORT) return;

            dmac.apply();
            if (outCtrl.outputPort == DROP_PORT) return;

            smac.apply();
    }
}

// deparser section
control TopDeparser(inout Parsed_packet p, packet_out b) {
    Checksum16() ck;
    apply {
        b.emit(p.ethernet);
        if (p.ip.isValid()) {
            ck.clear();                // prepare checksum unit
            p.ip.hdrChecksum = 16w0; // clear checksum
            ck.update(p.ip);          // compute new checksum.
            p.ip.hdrChecksum = ck.get();
        }
        b.emit(p.ip);
    }
}

// Instantiate the top-level VSS package
VSS(TopParser(),
    TopPipe(),
    TopDeparser()) main;
```

6. P4 language definition

The P4 language can be viewed as having several distinct components, which we describe separately:

- The core language, comprising of types, variables, scoping, declarations, statements, expressions, etc. We start by describing this part of the language.
- A sub-language for expressing parsers, based on state machines (Section 11).
- A sub-language for expressing computations using match-action units, based on traditional imperative control-flow (Section 12).
- A sub-language for describing architectures (Section 15).

6.1. Syntax and semantics

6.1.1. Grammar

The complete grammar of $P4_{16}$ is given in Appendix F, using Yacc/Bison grammar description language. This text is based on the same grammar. We adopt several standard conventions when we

provide excerpts from the grammar:

- UPPERCASE symbols denote terminals in the grammar.
- Excerpts from the grammar are given in BNF notation as follows:

```
p4program
   : /* empty */
   | p4program declaration
   | p4program ';'
   ;
```

Pseudo-code (mostly used for describing the semantics of various P4 constructs) are shown with fixed-size fonts as in the following example:

```
ParserModel.verify(bool condition, error err) {
     if (condition == false) {
          ParserModel.parserError = err;
          goto reject;
     }
}
```

6.1.2. Semantics and the P4 abstract machines

We describe the semantics of P4 in terms of abstract machines executing traditional imperative code. There is an abstract machine for each P4 sub-language (parser, control). The abstract machines are described in this text in pseudo-code and English.

P4 compilers are free to reorganize the code they generate in any way as long as the externally visible behaviors of the P4 programs are preserved as described by this specification where externally visible behavior is defined as:

- The input/output behavior of all P4 blocks, and
- The state maintained by extern blocks.

6.2. Preprocessing

To aid composition of programs from multiple source files P4 compilers should support the following subset of the C preprocessor functionality:

- `#define` for defining macros (without arguments)
- `#undef`
- `#if #else #endif #ifdef #ifndef #elif`
- `#include`

The preprocessor should also remove the sequence backslash newline (ASCII codes 92, 10) to facilitate splitting content across multiple lines when convenient for formatting.

Additional C preprocessor capabilities may be supported, but are not guaranteed—e.g., macros with arguments. Similar to C, `#include` can specify a file name either within double quotes or within <>.

```
# include <system_file>
# include "user_file"
```

The difference between the two forms is the order in which the preprocessor searches for header files when the path is incompletely specified.

P4 compilers should correctly handle `#line` directives that may be generated during preprocessing. This functionality allows P4 programs to be built from multiple source files, potentially produced by different programmers at different times:

- the P4 core library, defined in this document,
- the architecture, defining data plane interfaces and extern blocks,
- user-defined libraries of useful components (e.g. standard protocol header definitions), and
- the P4 programs that specify the behavior of each programmable block.

6.2.1. P4 core library

The P4 language specification defines a core library that includes several common programming constructs. A description of the core library is provided in Appendix B. All P4 programs must include the core library. Including the core library is done with

```
# include <core.p4>
```

6.3. Lexical constructs

All P4 keywords use only ASCII characters. All P4 identifiers must use only ASCII characters. P4 compilers should handle correctly strings containing 8-bit characters in comments and string literals. P4 is case-sensitive. Whitespace characters, including newlines are treated as token separators. Indentation is free-form; however, P4 has C-like block constructs, and all our examples use C-style indentation. Tab characters are treated as spaces.

The lexer recognizes the following kinds of terminals:

- `IDENTIFIER`: start with a letter or underscore, and contain letters, digits and underscores
- `TYPE`: identifier that denotes a type name
- `INTEGER`: integer literals
- `DONTCARE`: a single underscore
- Keywords such as `RETURN`. By convention, each keyword terminal corresponds to a language keyword with the same spelling but using lowercase. For example, the `RETURN` terminal corresponds to the `return` keyword.

6.3.1. Identifiers

P4 identifiers may contain only letters, numbers, and the underscore character _, and must start with a letter or underscore. The special identifier consisting of a single underscore _ is reserved to indicate a “don't care” value; its type may vary depending on the context. Certain keywords (e.g., `apply`) can be used as identifiers if the context makes it unambiguous.

```
nonTypeName
     : IDENTIFIER
     | APPLY
     | KEY
     | ACTIONS
     | STATE
     ;

name
     : nonTypeName
     | TYPE
     ;
```

6.3.2. Comments

P4 supports several kinds of comments:

- Single-line comments, introduced by `//` and spanning to the end of line,
- Multi-line comments, enclosed between `/*` and `*/`
- Nested multi-line comments are not supported.
- Javadoc-style comments, starting with `/**` and ending with `*/`

Use of Javadoc-style comments is strongly encouraged for the tables and actions that are used to synthesize the interface with the control-plane.

P4 treats comments as token separators and no comments are allowed within a token—e.g. `bi/**/t` is parsed as two tokens, `bi` and `t`, and not as a single token `bit`.

6.3.3. Literal constants

6.3.3.1. Boolean literals There are two Boolean literal constants: `true` and `false`.

6.3.3.2. Integer literals Integer literals are positive, arbitrary-precision integers. By default, literals are represented in base 10. To use a different base for the literal, one of the following prefixes must be employed:

- `0x` or `0X` indicates base 16 (hexadecimal)
- `0o` or `0O` indicates base 8 (octal)
- `0b` or `0B` indicates base 2

The width of a numeric literal in bits can be specified by an unsigned number prefix consisting of a number of bits and a signedness indicator:

- `w` indicates unsigned numbers
- `s` indicates signed numbers

Note that a leading zero by itself does not indicate an octal (base 8) constant. The underscore character is considered a digit within number literals but is ignored when computing the value of the parsed number. This allows long constant numbers to be more easily read by grouping digits together. The

underscore cannot be used in the width specification or as the first character of an integer literal. No comments or whitespaces are allowed within a literal. Here are some examples of numeric literals:

```
32w0xFF          // a 32-bit unsigned number with value 255
32s0xFF          // a 32-bit signed number with value 255
8w0b10101010    // an 8-bit unsigned number with value 0xAA
8w0b_1010_1010 // same value as above
8w170            // same value as above
8s0b1010_1010   // an 8-bit signed number with value -86
16w0377          // 16-bit unsigned number with value 377 (not 255!)
16w0o377         // 16-bit unsigned number with value 255 (base 8)
```

6.3.3.3. String literals String literals (string constants) are specified as an arbitrary sequence of 8-bit characters, enclosed within double quote signs " (ASCII code 34). Strings start with a double quote sign and extend to the first double quote sign which is not immediately preceded by an odd number of backslash characters (ASCII code 92). P4 does not make any validity checks on strings (i.e., it does not check that strings represent legal UTF-8 encodings).

Since P4 does not provide any operations on strings, string literals are generally passed unchanged through the P4 compiler to other third-party tools or compiler-backends, including the terminating quotes. These tools can define their own handling of escape sequences (e.g., how to specify Unicode characters, or handle unprintable ASCII characters).

Here are 3 examples of string literals:

```
"simple string"
"string \" with \" embedded \" quotes"
"string with embedded
line terminator"
```

6.4. Naming conventions

P4 provides a rich assortment of types. Base types include bit-strings, numbers, and errors. There are also built-in types for representing constructs such as parsers, pipelines, actions, and tables. Users can construct new types based on these: structures, enumerations, headers, header stacks, header unions, etc.

In this document we adopt the following conventions:

- Built-in types are written with lowercase characters—e.g., `int<20>`,
- User-defined types are capitalized—e.g., `IPv4Address`,
- Type variables are always uppercase—e.g., `parser P<H, IH>(...)`,
- Variables are uncapitalized— e.g., `ipv4header`,
- Constants are written with uppercase characters—e.g., `CPU_PORT`, and
- Errors and enumerations are written in camel-case— e.g. `PacketTooShort`.

6.5. P4 programs

A P4 program is a list of declarations:

```
p4program
     : /* empty */
     | p4program declaration
     | p4program ';'  /* empty declaration */
     ;

declaration
     : constantDeclaration
     | externDeclaration
     | actionDeclaration
     | parserDeclaration
     | typeDeclaration
     | controlDeclaration
     | instantiation
     | errorDeclaration
     | matchKindDeclaration
     ;
```

An empty declarations is indicated with a single semicolon. (Allowing empty declarations accommodates the habits of C/C++ and Java programmers—e.g., certain constructs, like `struct`, do not require a terminating semicolon).

6.5.1. Scopes

Some P4 constructs act as namespaces that create local scopes for names including:

- Derived type declarations (`struct`, `header`, `header_union`, `enum`), which introduce local scopes for field names,
- Block statements, which introduce local lexically-enclosed scopes,
- `parser`, `table`, `action`, and `control` blocks, which introduce local scopes
- Declarations with type variables, which introduce a new scope for those variables. For example, in the following `extern` declaration, the scope of the type variable `H` extends to the end of the declaration:

```
extern E<H>(...) { ... } // scope of H ends here.
```

The order of declarations is important; with the exception of parser states, all uses of a symbol must follow the symbol's declaration. (This is a departure from $P4_{14}$, which allows declarations in any order. This requirement significantly simplifies the implementation of compilers for P4, allowing compilers to use additional information about declared identifiers to resolve ambiguities.)

6.5.2. Stateful elements

Most P4 constructs are stateless: given some inputs they produce a result that solely depends on these inputs. There are only two stateful constructs that may retain information across packets:

- `table`s: Tables are read-only for the data plane, but their entries can be modified by the control-plane,
- `extern` objects: many objects have state that can be read and written by the control plane and data plane. All constructs from the $P4_{14}$ language version that encapsulate state (e.g., counters, meters, registers) are represented using `extern` objects in $P4_{16}$.

In P4 all stateful elements must be explicitly allocated at compilation-time through the process called "instantiation".

In addition, `parser`s, `control` blocks, and `package`s may contain stateful element instantiations. Thus, they are also treated as stateful elements, even if they appear to contain no state, and must be instantiated before they can be used. However, although they are stateful, `table`s do not need to be instantiated explicitly—declaring a `table` also creates an instance of it. This convention is designed to support the common case, since most tables are used just once. To have finer-grained control over when a `table` is instantiated, a programmer can declare it within a `control`.

Recall the example in Section 5.3: `TopParser`, `TopPipe`, `TopDeparser`, `Checksum16`, and `Switch` are types. There are two instances of `Checksum16`, one in `TopParser` and one in `TopDeparser`, both called `ck`. The `TopParser`, `TopDeparser`, `TopPipe`, and `Switch` are instantiated at the end of the program, in the declaration of the `main` instance object, which is an instance of the `Switch` type (a `package`).

6.6. L-values

L-values are expressions that may appear on the left side of an assignment operation or as arguments corresponding to `out` and `inout` function parameters. An l-value represents a storage reference. The following expressions are legal l-values:

```
prefixedNonTypeName
     : nonTypeName
     | dotPrefix nonTypeName
     ;

lvalue
     : prefixedNonTypeName
     | lvalue '.' member
     | lvalue '[' expression ']'
     | lvalue '[' expression ':' expression ']'
     ;
```

- Identifiers of a base or derived type.
- Structure, header, and header union field member access operations (using the dot notation).
- References to elements within header stacks (see Section 8.15): indexing, and references to `last` and `next`.
- The result of a bit-slice operator `[m:l]`.

The following is a legal l-value: `headers.stack[4].field`. Note that method and function calls cannot return l-values.

6.7. Calling convention: call by copy in/copy out

P4 provides multiple constructs for writing modular programs: extern methods, parsers, controls, actions. All these constructs behave similarly to procedures in standard general-purpose programming languages:

- They have named and typed parameters.
- They introduce a new local scope for parameters and local variables.
- They allow arguments to be passed by binding them to their parameters.

Invocations are executed using copy-in/copy-out semantics.

Each parameter may be labeled with a direction:

- `in` parameters are read-only. It is an error to use an `in` parameter on the left-hand side of an assignment or to pass it to a callee as a non-`in` argument. `in` parameters are initialized by copying the value of the corresponding argument when the invocation is executed.
- `out` parameters are uninitialized (parameters of type `header` or `header_union` are set to "invalid") and are treated as l-values (See Section 6.6) within the body of the method or function. An arguments passed as an `out` parameter must be an l-value; after the execution of the call, the value of the parameter is copied to the corresponding storage location for that l-value.
- `inout` parameters are both `in` and `out`. An argument passed as an `inout` parameter must be an l-value.
- No direction indicates that value of parameter is either:
  - a compile-time known value
  - an action parameter that can only be set by the control plane
  - an action parameter that can be set directly by another calling action; in this case it behaves like an `in` parameter

Arguments are evaluated from left to right prior to the invocation of the function itself. The order of evaluation is important when the expression supplied for an argument can have side-effects. Consider the following example:

```
extern void f(inout bit x, in bit y);
extern bit g(inout bit z);
bit a;
f(a, g(a));
```

Note that the evaluation of `g` may mutate its argument `a`, so the compiler has to ensure that the value passed to `f` for its first parameter is not changed by the evaluation of the second argument. The semantics for evaluating a function call is given by the following algorithm (implementations can be different as long as they provide the same result):

1. Arguments are evaluated from left to right as they appear in the function call expression.

2. For each `out` and `inout` argument the corresponding l-value is saved (so it cannot be changed by the evaluation of the following arguments). This is important if the argument contains indexing operations into a header stack.
3. The value of each argument is saved into a temporary.
4. The function is invoked with the temporaries as arguments. We are guaranteed that the temporaries that are passed as arguments are never aliased to each other, so this "generated" function call can be implemented using call-by-reference if supported by the architecture.
5. On function return, the temporaries that correspond to `out` or `inout` arguments are copied in order from left to right into the l-values saved in step 2.

According to this algorithm, the previous function call is equivalent to the following sequence of statements:

```
bit tmp1 = a;      // evaluate a; save result
bit tmp2 = g(a);   // evaluate g(a); save result; modifies a
f(tmp1, tmp2);     // evaluate f; modifies tmp1
a = tmp1;          // copy inout result back into a
```

To see why Step 2 in the above algorithm is important, consider the following example:

```
header H { bit z; }
H[2] s;
f(s[a].z, g(a));
```

The evaluation of this call is equivalent to the following sequence of statements:

```
bit tmp1 = a;            // save the value of a
bit tmp2 = s[tmp1].z;   // evaluate first argument
bit tmp3 = g(a);         // evaluate second argument; modifies a
f(tmp2, tmp3);           // evaluate f; modifies tmp2
s[tmp1].z = tmp2;        // copy inout result back; dest is not s[a].z
```

When used as arguments, `extern` objects can only be passed as directionless parameters—e.g., see the packet argument in the very simple switch example.

6.7.1. Justification

The main reason for using copy-in/copy-out semantics (instead of the more common call-by-reference semantics) is for controlling the side-effects of `extern` functions and methods. `extern` methods and functions are the main mechanism by which a P4 program communicates with its environment. With copy-in/copy-out semantics `extern` functions cannot hold references to P4 program objects; this enables the compiler to limit the side-effects that `extern` functions may have on the P4 program both in space (they can only affect `out` parameters) and in time (side-effects can only occur at function call time).

In general, `extern` functions are arbitrarily powerful: they can store information in global storage, spawn separate threads, "collude" with each other to share information — but they cannot access any variable in a P4 program. With copy-in/copy-out semantics the compiler can still reason about P4 programs that invoke `extern` functions.

There are additional benefits of using copy-in copy-out semantics:

- It enables P4 to be compiled for architectures that do not support references (e.g., where all data is allocated to named registers. Such architectures may require indices into header stacks that appear in a program to be compile-time known values.)
- It simplifies some compiler analyses, since function parameters can never alias to each other within the function body.

```
parameterList
     : /* empty */
     | nonEmptyParameterList
     ;

nonEmptyParameterList
     : parameter
     | nonEmptyParameterList ',' parameter
     ;

parameter
     : optAnnotations direction typeRef name
     ;

direction
     : IN
     | OUT
     | INOUT
     | /* empty */
     ;
```

Following is a summary of the constraints imposed by the parameter directions:

- When used as arguments, extern objects can only be passed as directionless parameters.
- All constructor parameters are evaluated at compilation-time, and in consequence they must all be directionless (they cannot be `in`, `out`, or `inout`); this applies to `package`, `control`, `parser`, and `extern` objects. Values for these parameters must be specified at compile-time, and must evaluate to compile-time known values. See Section 13 for further details.
- For actions all directionless parameters must be at the end of the parameter list. When an action appears in a `table`'s `actions` list, only the parameters with a direction must be bound. See Section 12.1 for further details.
- Actions can also be explicitly invoked using function call syntax, either from a control block or from another action. In this case, values for all action parameters must be supplied explicitly, including values for the directionless parameters. The directionless parameters in this case behave like `in` parameters. See Section 12.1.1 for further details.

6.8. Name resolution

P4 objects that introduce namespaces are organized in a hierarchical fashion. There is a top-level unnamed namespace containing all top-level declarations.

Identifiers prefixed with a dot are always resolved in the top-level namespace.

```
const bit<32> x = 2;
control c() {
    int<32> x = 0;
    apply {
        x = x + (int<32>).x;  // x is the int<32> local variable,
                              // .x is the top-level bit<32> variable
    }
}
```

References to resolve an identifier are attempted inside-out, starting with the current scope and proceeding to all lexically enclosing scopes. The compiler may provide a warning if multiple resolutions are possible for the same name (name shadowing).

```
const bit<4> x = 1;
control p() {
    const bit<8> x = 8;     // x declaration shadows global x
    const bit<4> y = .x;    // reference to top-level x
    const bit<8> z = x;     // reference to p's local x
    apply {}
}
```

6.9. Visibility

Identifiers defined in the top-level namespace are globally visible. Declarations within a `parser` or `control` are private and cannot be referred to from outside of the enclosing `parser` or `control`.

7. P4 data types

$P4_{16}$ is a statically-typed language. Programs that do not pass the type checker are considered invalid and rejected by the compiler. P4 provides a number of base types as well as type operators that construct derived types. Some values can be converted to a different type using casts. However, to make user intents clear, implicit casts are only allowed in a few circumstances and the range of casts available is intentionally restricted.

7.1. Base types

P4 supports the following built-in base types:

- The `void` type, which has no values and can be used only in a few restricted circumstances.
- The `error` type, which is used to convey errors in a target-independent, compiler-managed way.

- The `match_kind` type, which is used for describing the implementation of table lookups,
- `bool`, which represents Boolean values
- Bit-strings of fixed width, denoted by `bit<>`
- Fixed-width signed integers represented using two's complement `int<>`
- Bit-strings of dynamically-computed width with a fixed maximum width `varbit<>`

```
baseType
    : BOOL
    | ERROR
    | BIT
    | BIT '<' INTEGER '>'
    | INT '<' INTEGER '>'
    | VARBIT '<' INTEGER '>'
    ;
```

7.1.1. The void type

The void type is written `void`. It contains no values. It is not included in the production rule `baseType` as it can only appear in few restricted places in P4 programs.

7.1.2. The error type

The error type contains opaque values that can be used to signal errors. It is written as `error`. New constants of the error type are defined with the syntax:

```
errorDeclaration
    : ERROR '{' identifierList '}'
    ;
```

All `error` constants are inserted into the `error` namespace, irrespective of the place where an error is defined. `error` is similar to an enumeration (`enum`) type in other languages. A program can contain multiple `error` declarations, which the compiler will merge together. It is an error to declare the same identifier multiple times. Expressions of type `error` are described in Section 8.2.

For example, the following declaration creates two constants of `error` type (these errors are declared in the P4 core library):

```
error { ParseError, PacketTooShort }
```

The underlying representation of errors is target-dependent.

7.1.3. The match kind type

The `match_kind` type is very similar to the `error` type and is used to declare a set of names that may be used in a table's key property (described in Section 12.2.1). All identifiers are inserted into the top-level namespace. It is an error to declare the same `match_kind` identifier multiple times.

```
matchKindDeclaration
    : MATCH_KIND '{' identifierList '}'
    ;
```

The P4 core library contains the following match_kind declaration:

```
match_kind {
    exact,
    ternary,
    lpm
}
```

Architectures may support additional `match_kind`s. The declaration of new `match_kind`s can only occur within model description files; P4 programmers cannot declare new match kinds.

7.1.4. The Boolean type

The Boolean type `bool` contains just two values, `false` and `true`. Boolean values are not integers or bit-strings.

7.1.5. Strings

P4 offers no support for string processing. The only strings that can appear in a P4 program are constant string literals, described in Section 6.3.3.3. String literals can only be used in annotations (described in Section 17). For example, the following annotation indicates that a specific name should be used for a table when generating the control-plane API:

```
@name("acl") table t1 { ...}
```

7.1.6. Integers (signed and unsigned)

P4 supports arbitrary-size integer values. The typing rules for the integer types are chosen according to the following principles:

- Inspired by C: Typing of integers is modeled after the well-defined parts of C, expanded to cope with arbitrary fixed-width integers. In particular, the type of the result of an expression only depends on the expression operands, and not on how the result of the expression is consumed.
- No undefined behaviors: P4 attempts to avoid many of C's behaviors, which include the size of an integer (int), the results produced on overflow, and the results produced for some input combinations (e.g., shifts with negative amounts, overflows on signed numbers, etc.). P4 computations on integer types have no undefined behaviors.
- Least surprise: The P4 typing rules are chosen to behave as closely as possible to traditional well-behaved C programs.
- Forbid rather than surprise: Rather than provide surprising or undefined results (e.g., in C comparisons between signed and unsigned integers), we have chosen to forbid expressions with ambiguous interpretations. For example, P4 does not allow binary operations that combine signed and unsigned integers.

The priority of arithmetic operations is identical to C—e.g., multiplication binds tighter than addition.

7.1.6.1. Portability

No P4 target can support all possible types and operations. For example, the type `bit<23132312>` is legal in P4, but it is highly unlikely to be supported on any target in practice. Hence, each target can impose restrictions on the types it can support. Such restrictions may include:

- The maximum width supported
- Alignment and padding constraints (e.g., arithmetic may only be supported on widths which are an integral number of bytes).
- Constraints on some operands (e.g., some architectures may only support multiplications with small constants, or shifts with small values).

The documentation supplied with a target should clearly specify restrictions, and target-specific compilers should provide clear error messages when such restrictions are encountered. An architecture may reject a well-typed P4 program and still be conformant to the P4 spec. However, if an architecture accepts a P4 program as valid, the runtime program behavior should match this specification.

7.1.6.2. Unsigned integers (bit-strings)

An unsigned integer (which we also call a "bit-string") has an arbitrary width, expressed in bits. A bit-string of width `W` is declared as: `bit<W>`. `W` must be a compile-time known value (see Section 16.1) that evaluates to a positive integer greater than 0.

Bits within a bit-string are numbered from `0` to `W-1`. Bit `0` is the least significant, and bit `W-1` is the most significant.

For example, the type `bit<128>` denotes the type of bit-string values with 128 bits numbered from 0 to 127, where bit 127 is the most significant.

The type `bit` is a shorthand for `bit<1>`.

P4 architectures may impose additional constraints on bit types: for example, they may limit the maximum size, or they may only support some arithmetic operations on certain sizes (e.g., 16-, 32-, and 64- bit values).

All operations that can be performed on unsigned integers are described in Section 8.5.

7.1.6.3. Signed Integers

Signed integers are represented using two's complement. An integer with `W` bits is declared as: `int<W>`. `W` must be a compile-time known value evaluating to a positive integer greater than 1.

Bits within an integer are numbered from `0` to `W-1`. Bit `0` is the least significant, and bit `W-1` is the sign bit.

For example, the type `int<64>` describes the type of integers represented using exactly 64 bits with bits numbered from 0 to 63, where bit 63 is the most significant (sign) bit.

P4 architectures may impose additional constraints on signed types: for example, they may limit the maximum size, or they may only support some arithmetic operations on certain sizes (e.g., 16-, 32-, and 64- bit values).

All operations that can be performed on signed integers are described in Section 8.6.

7.1.6.4. Dynamically-sized bit-strings

Some network protocols use fields whose size is only known at runtime (e.g., IPv4 options). To support restricted manipulations of such values, P4 provides a special bit-string type whose size is set at runtime, called a `varbit`.

The type varbit<W> denotes a bit-string with a width of at most W bits, where W must be a positive integer that is a compile-time known value. For example, the type varbit<120> denotes the type of bit-string values that may have between 0 and 120 bits. Most operations that are applicable to fixed-size bit-strings (unsigned numbers) cannot be performed on dynamically sized bit-strings.

P4 architectures may impose additional constraints on varbit types: for example, they may limit the maximum size, or they may require varbit values to always contain an integer number of bytes at runtime.

All operations that can be performed on varbits are described in Section 8.8.

7.1.6.5. Infinite-precision integers

The infinite-precision data type describes integers with an unlimited precision. This type is written as int.

This type is reserved for integer literals and expressions that involve only literals. No P4 runtime value can have an int type; at compile time the compiler will convert all int values that have a runtime component to fixed-width types, according to the rules described below.

All operations that can be performed on infinite-precision integers are described in Section 8.7.

7.1.6.6. Integer literal types

The types of integer literals (constants) are as follows:

- A simple integer constant has type int.
- A positive integer prefixed with an integer width N and the character w has type bit<N>.
- An integer prefixed with an integer width N and the character s has type int<N>.

The table below shows several examples of integer literals and their types. For additional examples of literals see Section 6.3.3.

Literal	Interpretation
10	Type is int, value is 10
8w10	Type is bit<8>, value is 10
8s10	Type is int<10>, value is 10
2s3	Type is int<2>, value is -1 (last 2 bits), overflow warning
1w10	Type is bit<1>, value is 0 (last bit), overflow warning
1s10	Error: 1-bit signed type is illegal

7.2. Derived types

P4 provides a number of type constructors that can be used to derive additional types including:

- enum
- header
- header stacks
- struct
- header_union
- tuple
- type specialization
- extern
- parser
- control
- package

The types `header`, `header_union`, `enum`, `struct`, `extern`, `parser`, `control`, and `package` can only be used in type declarations, where they introduce a new name for the type. The type can subsequently be referred to using this identifier.

Other types cannot be declared, but are synthesized by the compiler internally to represent the type of certain language constructs. These types are described in Section 7.2.8: set types and function types. For example, the programmer cannot declare a variable with type "set", but she can write an expression whose value evaluates to a `set` type. These types are used during type-checking.

```
typeDeclaration
     : derivedTypeDeclaration
     | typedefDeclaration
     | parserTypeDeclaration ';'
     | controlTypeDeclaration ';'
     | packageTypeDeclaration ';'
     ;

derivedTypeDeclaration
     : headerTypeDeclaration
     | headerUnionDeclaration
     | structTypeDeclaration
     | enumDeclaration
     ;

typeRef
     : baseType
     | typeName
     | specializedType
     | headerStackType
     | tupleType
     ;

prefixedType
     : TYPE
     | dotPrefix TYPE
     ;

typeName
     : prefixedType
     ;
```

7.2.1. Enumeration types

An enumeration type is defined using the following syntax:

```
enumDeclaration
     : optAnnotations ENUM name '{' identifierList '}'
     ;

identifierList
     : name
     | identifierList ',' name
     ;
```

For example, the declaration

```
enum Suits { Clubs, Diamonds, Hearths, Spades }
```

introduces a new enumeration type, which contains four constants—e.g., `Suits.Clubs`. An `enum` declaration introduces a new identifier in the current scope for naming the created type. The underlying representation of such values is not specified, so their "size" in bits is not specified (it is target-specific).

Annotations, represented by the non-terminal `optAnnotations`, are described in Section 17.

Operations on `enum` values are described in Section 8.3.

7.2.2. Header types

The declaration of a `header` type is given by the following syntax:

```
headerTypeDeclaration
     : optAnnotations HEADER name '{' structFieldList '}'
     ;

structFieldList
     : /* empty */
     | structFieldList structField
     ;

structField
     : optAnnotations typeRef name ';'
     ;
```

where each `typeRef` is restricted to a bit-string type (fixed or variable) or an integer type. This declaration introduces a new identifier in the current scope; the type can be referred to using this identifier. A header is similar to a `struct` in C, containing all the specified fields. However, in addition, a header also contains a hidden Boolean "validity" field. When the "validity" bit is `true` we say that the "header is valid". When a header is created its "validity" bit is automatically set to `false`. The "validity" bit can be manipulated by using the header methods `isValid()`, `setValid()`, and `setInvalid()`, as described in Section 8.14.

Header types may be empty:

```
header Empty_h { }
```

Note that an empty header still contains a validity bit.

Headers that do not contain any `varbit` field are "fixed size." Headers containing `varbit` fields have "variable size." The size (in bits) of a fixed-size header is a constant, and it is simply the sum of the sizes of all component fields (without counting the validity bit). There is no padding or alignment of the header fields. Architectures may impose additional constraints on header types—e.g., restricting headers to sizes that are an integer number of bytes.

For example, the following declaration describes a typical Ethernet header:

```
header Ethernet_h {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}
```

The following variable declaration uses the newly introduced type `Ethernet_h`:

```
Ethernet_h ethernetHeader;
```

P4's parser language provides an `extract` method that can be used to "fill in" the fields of a header from a network packet, as described in Section 11.8. The successful execution of an `extract` operation also sets the validity bit of the extracted header to `true`.

Here is an example of an IPv4 header with variable-sized options:

```
header IPv4_h {
    bit<4>        version;
    bit<4>        ihl;
    bit<8>        diffserv;
    bit<16>       totalLen;
    bit<16>       identification;
    bit<3>        flags;
    bit<13>       fragOffset;
    bit<8>        ttl;
    bit<8>        protocol;
    bit<16>       hdrChecksum;
    bit<32>       srcAddr;
    bit<32>       dstAddr;
    varbit<320>   options;
}
```

As discussed in Section 8.11, headers that contain variable-length fields may need to be parsed in multiple steps by being broken into multiple headers.

7.2.3. Header stacks

A header stack represents an array of headers. A header stack type is defined as:

```
headerStackType
     : typeName '[' expression ']'
     ;
```

where `typeName` is the name of a header type. For a header stack `hs[n]`, the term `n` is the maximum defined size, and must be a positive integer that is a compile-time known value. Nested header stacks are not supported. At runtime a stack contains `n` values with type `typeName`, only some of which may be valid. Expressions on header stacks are discussed in Section 8.15.

For example, the following declarations,

```
header Mpls_h {
     bit<20> label;
     bit<3>  tc;
     bit     bos;
     bit<8>  ttl;
}
Mpls_h[10] mpls;
```

introduce a header stack called `mpls` containing ten entries, each of type `Mpls_h`.

7.2.4. Header unions

A header union represents an alternative containing at most one of several different headers. Header unions can be used to represent "options" in protocols like TCP and IP. They also provide hints to P4 compilers that only one alternative will be present, allowing them to conserve storage resources.

A header union is defined as:

```
headerUnionDeclaration
     : optAnnotations HEADER_UNION name
       '{' structFieldList '}'
     ;
```

This declaration introduces a new type with the specified name in the current scope. Each element of the list of fields used to declare a header union must be a header type. However, the empty list of fields is legal.

As an example, the type `Ip_h` below represents the union of an IPv4 and IPv6 headers:

```
header_union IP_h {
   IPv4_h v4;
   IPv6_h v6;
}
```

A header union is not considered a type with fixed width.

7.2.5. Struct types

P4 `struct` types are defined with the following syntax:

```
structTypeDeclaration
     : optAnnotations STRUCT name '{' structFieldList '}'
     ;
```

This declaration introduces a new type with the specified name in the current scope. An empty struct is legal. For example, the structure `Parsed_headers` below contains the headers recognized by a simple parser:

```
header Tcp_h { ... }
header Udp_h { ... }
struct Parsed_headers {
     Ethernet_h ethernet;
     Ip_h       ip;
     Tcp_h      tcp;
     Udp_h      udp;
}
```

7.2.6. Tuple types

A tuple is similar to a `struct`, in that it holds multiple values. Unlike a `struct` type, tuples have no named fields. The type of tuples with n component types `T1`,...,`Tn` is written as

```
tuple<T1,...,Tn>
```

```
tupleType
     : TUPLE '<' typeArgumentList '>'
     ;
```

Operations that manipulate tuple types are described in Sections 8.10 and 8.12.

7.2.7. Type nesting rules

The table below lists all types that may appear as members of headers, header unions, structs, and tuples. Note that `int` means an infinite-precision integer, without a width specified.

Element type	Container type		
	header	header_union	struct or tuple
bit<W>	allowed	error	allowed
int<W>	allowed	error	allowed
varbit<W>	allowed	error	allowed
int	error	error	error
void	error	error	error
error	error	error	allowed
match_kind	error	error	error
bool	error	error	allowed
enum	error	error	allowed
header	error	allowed	allowed
header stack	error	error	allowed
header_union	error	error	allowed
struct	error	error	allowed
tuple	error	error	allowed

Rationale: `int` does not have precise storage requirements, unlike `bit<>` or `int<>` types. `match_kind` values are not useful to store in a variable, as they are only used to specify how to match fields in table search keys, which are all declared at compile time. `void` is not useful as part of another data structure. Headers must have precisely defined formats as sequences of bits in order for them to be parsed or deparsed.

Note the two-argument `extract` method (see Section 11.8.2) on packets only supports a single `varbit` field in a header.

7.2.8. Synthesized data types

For the purposes of type-checking the P4 compiler can synthesize some type representations which cannot be directly expressed by users. These are described in this section: set types and function types.

7.2.8.1. Set types The type `set<T>` describes sets of values of type `T`. Set types can only appear in restricted contexts in P4 programs. For example, the range expression `8w5 .. 8w8` describes a set containing the 8-bit numbers 5, 6, 7, and 8, so its type is `set<bit<8>>;`. This expression can be used as a label in a `select` expression (see Section 11.6), matching any value in this range. Set types cannot be named or declared by P4 programmers, they are only synthesized by the compiler internally and used for type-checking. Expressions with set types are described in Section 8.12.

7.2.8.2. Function types Currently function types cannot be created explicitly in P4 programs; they are created by the P4 compiler internally to represent the types of functions, procedures, and methods during type-checking. We also call the type of a function its signature. Libraries can contain extern function declarations.

For example, the following declaration:

```
extern void random(in bit<5> logRange, out bit<32> value);
```

describes an object `random` which has a function type, representing the following information:

- the result type is `void`
- the function has two inputs
- first input has direction `in`, type `bit<5>`, and name `logRange`
- second input has direction `out`, type `bit<32>`, and name `value`

7.2.9. Extern types

P4 supports extern object declarations and extern function declarations using the following syntax.

```
externDeclaration
    : optAnnotations EXTERN nonTypeName optTypeParameters '{' methodPrototypes '}'
    | optAnnotations EXTERN functionPrototype ';'
    ;
```

7.2.9.1. Extern functions An extern function declaration describes the name and type signature of the function, but not its implementation.

```
functionPrototype
    : typeOrVoid name optTypeParameters '(' parameterList ')'
    ;
```

For an example of an `extern` function declaration, see Section 7.2.8.2.

7.2.9.2. Extern objects An extern object declaration declares an object and all methods that can be invoked to perform computations and to alter the state of the object. Extern object declarations can also optionally declare constructor methods; these must have the same name as the enclosing `extern` type, no type parameters, and no return type. Extern declarations may only appear as allowed by the architecture model and may be specific to a target.

```
methodPrototypes
    : /* empty */
    | methodPrototypes methodPrototype
    ;

methodPrototype
    : optAnnotations functionPrototype ';'
    | optAnnotations TYPE '(' parameterList ')' ';' //constructor
    ;

typeOrVoid
    : typeRef
    | VOID
    | nonTypeName      // may be a type variable
    ;
```

```
optTypeParameters
     : /* empty */
     | '<' typeParameterList '>'
     ;

typeParameterList
     : nonTypeName
     | typeParameterList ',' nonTypeName
     ;
```

For example, the P4 core library introduces two extern objects `packet_in` and `packet_out` used for manipulating packets (see Sections 11.8 and 14). Here is an example showing how the methods of these objects can be invoked on a packet:

```
extern packet_out {
     void emit<T>(in T hdr);
}
control d(packet_out b, in Hdr h) {
     apply {
          b.emit(h.ipv4);        // write ipv4 header into output packet
     }                             // by calling emit method
}
```

Functions and methods are the only P4 constructs that support overloading: there can exist multiple methods with the same name in the same scope. Even so, two functions (or methods of an `extern` object) can have the same name only if they have a different number of parameters.

7.2.10. Type specialization

A generic type may be specialized by specifying arguments for its type variables. In cases where the compiler can infer type arguments type specialization is not necessary. When a type is specialized all its type variables must be bound.

```
specializedType
     : prefixedType '<' typeArgumentList '>'
     ;
```

For example, the following extern declaration describes a generic block of registers, where the type of the elements stored in each register is an arbitrary `T`.

```
extern Register<T> {
     Register(bit<32> size);
     T read(bit<32> index);
     void write(bit<32> index, T value);
}
```

The type `T` has to be specified when instantiating a set of registers, by specializing the Register type:

```
Register<bit<32>>(128) registerBank;
```

The instantiation of `registerBank` is made using the `Register` type specialized with the `bit<32>` bound to the `T` type argument.

7.2.11. Parser and control blocks types

Parsers and control blocks types are similar to function types: they describe the signature of parsers and control blocks. Such functions have no return values. Declarations of parsers and control block types in architectures may be generic (i.e., have type parameters).

The types `parser`, `control`, and `package` cannot be used as types of arguments for methods, parsers, controls, tables, actions. They can be used as types for the arguments passed to constructors (see Section 13).

7.2.11.1. Parser type declarations A parser type declaration describes the signature of a parser. A parser should have at least one argument of type `packet_in`, representing the received packet that is processed.

```
parserTypeDeclaration
    : optAnnotations PARSER name optTypeParameters
     '(' parameterList ')'
    ;
```

For example, the following is a type declaration of a parser type named `P` that is parameterized on a type variable `H`. The parser that receives as input a `packet_in` value `b` and produces two values:

- A value with a user-defined type `H`
- A value with a predefined type `Counters`

```
struct Counters { ... }
parser P<H>(packet_in b,
            out H packetHeaders,
            out Counters counters);
```

7.2.11.2. Control type declarations A control type declaration describes the signature of a control block.

```
controlTypeDeclaration
    : optAnnotations CONTROL name optTypeParameters
      '(' parameterList ')'
    ;
```

Control type declarations are similar to parser type declarations.

7.2.12. Package types

A package type describes the signature of a package.

```
packageTypeDeclaration
     : optAnnotations PACKAGE name optTypeParameters
       '(' parameterList ')'
     ;
```

All parameters of a package are evaluated at compilation-time, and in consequence they must all be directionless (they cannot be `in`, `out`, or `inout`). Otherwise package types are very similar to parser type declarations. Packages can only be instantiated; there are no runtime behaviors associated with them.

7.2.13. Don't care types

A don't care (underscore, "_") can be used in some circumstances as a type. It should be only used in a position where one could write a bound type variable. The underscore can be used to reduce code complexity—when it is not important what the type variable binds to (during type unification the don't care type can unify with any other type). An example is given Section 15.1.

7.3. typedef

A `typedef` declaration can be used to give an alternative name to a type.

```
typedefDeclaration
     : optAnnotations TYPEDEF typeRef name ';'
     | optAnnotations TYPEDEF derivedTypeDeclaration name ';'
     ;
```

```
typedef bit<32> u32;
typedef struct Point { int<32> x; int<32> y; } Pt;
typedef Empty_h[32] HeaderStack;
```

The two types are treated as synonyms, and all operations that can be executed using the original type can be also executed using the newly created type.

8. Expressions

This section describes all expressions that can be used in P4, grouped by the type of value they produce.

The grammar production rule for general expressions is as follows:

```
expression
     : INTEGER
     | TRUE
     | FALSE
     | STRING_LITERAL
```

```
    | nonTypeName
    | '.' nonTypeName
    | expression '[' expression ']'
    | expression '[' expression ':' expression ']'
    | '{' expressionList '}'
    | '(' expression ')'
    | '!' expression
    | '~' expression
    | '-' expression
    | '+' expression
    | typeName '.' member
    | ERROR '.' member
    | expression '.' member
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | expression '+' expression
    | expression '-' expression
    | expression SHL expression       // SHL is <<
    | expression '>''>' expression    // check that >> are contiguous
    | expression LE expression        // LE is <=
    | expression GE expression
    | expression '<' expression
    | expression '>' expression
    | expression NE expression        // NE is !=
    | expression EQ expression        // EQ is ==
    | expression '&' expression
    | expression '^' expression
    | expression '|' expression
    | expression PP expression        // PP is ++
    | expression AND expression       // AND is &&
    | expression OR expression        // OR is ||
    | expression '?' expression ':' expression
    | expression '<' typeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'
    | typeRef '(' argumentList ')'
    | '(' typeRef ')' expression
    ;

expressionList
    : /* empty */
    | expression
    | expressionList ',' expression
    ;

member
```

```
    : name
    ;

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList ',' argument
    ;

argument
    : expression
    ;

typeArg
    : DONTCARE
    | typeRef
    ;

typeArgumentList
    : typeArg
    | typeArgumentList ',' typeArg
    ;
```

See Appendix F for the complete P4 grammar.

This grammar does not indicate the precedence of the various operators. The precedence follows exactly the C precedence rules. Concatenation (`++`) has the same precedence as infix addition. Bit-slicing `a[m:l]` has the same precedence as array indexing (`a[i]`). An additional semantic check is required for right shift to check that there is no space between the two consecutive greater-than signs `>>`. This rule is required to allow parsing for both the right shift operators and specialized types, such as in `function<bit<32>>`.

In addition to these expressions, P4 also supports `select` expressions (described in Section 11.6), which may be used only in parsers.

8.1. Expression evaluation order

Given a compound expression, the order in which sub-expressions are evaluated is important when the sub-expressions have side-effects. P4 expressions are evaluated as follows:

- Boolean operators `&&` and `||` use short-circuit evaluation—i.e., the second operand is only evaluated if necessary.
- The conditional operator `e1 ? e2 : e3` evaluates `e1`, and then either evaluates `e2` or `e3`.
- All other expressions are evaluated left-to-right as they appear in the source program.

- Method and function calls are evaluated as described in Section 6.7.

8.2. Operations on `error` types

Symbolic names declared by an `error` declaration belong to the `error` namespace. The `error` type only supports equality (`==`) and inequality (`!=`) comparisons. The result of such a comparison is a Boolean value.

For example, the following operation tests for the occurrence of an error:

```
error errorFromParser;
...
if (errorFromParser != error.NoError) { ... }
```

8.3. Operations on `enum` types

Symbolic names declared by an enum belong to the namespace introduced by the enum declaration rather than the top-level namespace.

```
enum X { v1, v2, v3 }
X.v1  // reference to v1
v1    // error - v1 is not in the top-level namespace
```

Similar to errors, `enum` expressions only support equality (`==`) and inequality (`!=`) comparisons. Expressions whose type is an `enum` cannot be cast to or from any other type.

Note that if an `enum` value appears in the control-plane API, the compiler must select a suitable serialization data type and representation.

8.4. Expressions on Booleans

The following operations are provided on Boolean expressions: - And, denoted by `&&`, - Or denoted by `||`, - Negation, denoted by `!`, and - Equality and inequality tests, denoted by `==` and `!=` respectively.

The precedence of these operators is similar to C and uses short-circuit evaluation.

P4 does not implicitly cast from bit-strings to Booleans or vice versa. As a consequence, a program that is valid in a language like C such as,

```
if (x) ...
```

(where x has an integer type) must instead be written in P4 as:

```
if (x != 0) ...
```

See the discussion on infinite-precision types and implicit casts in Section 8.9.2 for details on how the `0` in this expression is evaluated.

8.4.1. Conditional operator

A conditional expression of the form `e1 ? e2 : e3` behaves the same as in languages like C. As described above, the expression `e1` is evaluated first, and either `e2` or `e3` is evaluated depending on the result.

The first sub-expression `e1` must have type Boolean, and the second and third sub-expressions must have the same type, which cannot both be infinite precision integers unless the condition itself can be evaluated at compilation time. This restriction is designed to ensure that the width of the result of the conditional expression can be inferred statically at compile time.

8.5. Operations on bit types (unsigned integers)

This section discusses all operations that can be performed on expressions of type `bit<W>` for some width `W`, also known as bit-strings.

Arithmetic operations "wrap-around", similar to C operations on unsigned values (i.e., representing a large value on W bits will only keep the least-significant W bits of the value). In particular, P4 does not have arithmetic exceptions—the result of an arithmetic operation is defined for all possible inputs.

All binary operations (except shifts) require both operands to have the same exact type and width; supplying operands with different widths produces an error at compile time. No implicit casts are inserted by the compiler to equalize the widths. There are no binary operations that combine signed and unsigned values (except shifts). The following operations are provided on bit-string expressions:

- Test for equality between bit-strings of the same width, designated by `==`. The result is a Boolean value.
- Test for inequality between bit-strings of the same width, designated by `!=`. The result is a Boolean value.
- Unsigned comparisons `<,>,<=,>=`. Both operands must have the same width and the result is a Boolean value.

Each of the following operations produces a bit-string result when applied to bit-strings of the same width:

- Negation, denoted by unary `-`. The result is computed by subtracting the value from 2^W. The result is unsigned and has the same width as the input. The semantics is the same as the C negation of unsigned numbers.
- Unary plus, denoted by `+`. This operation behaves like a no-op.
- Addition, denoted by `+`. This operation is associative. The result is computed by truncating the result of the addition to the width of the output (similar to C).
- Subtraction, denoted by `-`. The result is unsigned, and has the same type as the operands. It is computed by adding the negation of the second operand (similar to C).
- Multiplication, denoted by `*`. The result has the same width as the operands and is computed by truncating the result to the output's width. P4 architectures may impose additional restrictions—e.g., they may only allow multiplication by a power of two.
- Bitwise "and" between two bit-strings of the same width, denoted by `&`.
- Bitwise "or" between two bit-strings of the same width, denoted by `|`.
- Bitwise "complement" of a single bit-string, denoted by `~`.
- Bitwise "xor" of two bit-strings of the same width, denoted by `^`.

Bit-strings also support the following operations:

- Concatenation of two bit-strings, denoted by the infix operator ++. The result is a bit-string whose length is the sum of the lengths of the inputs where the most significant bits are taken from the left operand.
- Extraction of a set of contiguous bits, also known as a slice, denoted by `[m:l]`, where `m` and `l` must be positive integers that are compile-time known values, and `m >= l`. The result is a bit-string of width `m - l + 1`, including the bits numbered from `l` (which becomes the least significant bit of the result) to `m` (the most significant bit of the result) from the source operand. The conditions `0 <= l < W` and `l <= m < W` are checked statically (where `W` is the length of the source bit-string). Note that both endpoints of the extraction are inclusive. The bounds are required to be compile-time known values so that the result width can be computed at compile time. Slices are also l-values, which means that P4 supports assigning to a slice: `e[m:l] = x` . The effect of this statement is to set bits `m` to `l` of `e` to the bit-pattern represented by `x`, and leaves all other bits of `e` unchanged.
- Logical shift left and right with a runtime known unsigned integer value, denoted by << and >> respectively. In a shift, the left operand is unsigned, and right operand must be either an expression of type `bit<S>` or a non-negative integer literal. The result has the same type as the left operand. Shifting by an amount greater than the width of the input produces a result where all bits are zero.

8.6. Operations on fixed-width signed integers

This section discusses all operations that can be performed on expressions of type `int<W>` for some `W`. Recall that the `int<W>` denotes signed `W`-bit integers, represented using two's complement.

In general, P4 arithmetic operations do not detect "underflow" or "overflow": operations simply "wrap around", similar to C operations on unsigned values. Hence, attempting to represent large values using `W` bits will only keep the least-significant `W` bits of the value.

P4 also does not support arithmetic exceptions. The runtime result of an arithmetic operation is defined for all combinations of input arguments.

All binary operations (except shifts) require both operands to have the same exact type (signedness) and width and supplying operands with different widths or signedness produces a compile-time error. No implicit casts are inserted by the compiler to equalize the types. With the exception of shifts, P4 does not have any binary operations that combine signed and unsigned values.

Note that bitwise operations on signed integers are well-defined, since the representation is mandated to be two's complement.

The `int<W>` datatype supports the following operations; all binary operations require both operands to have the exact same type. The result always has the same width as the left operand.

- Negation, denoted by unary –.
- Unary plus, denoted by +. This operation behaves like a no-op.
- Addition, denoted by +.
- Subtraction, denoted by –.
- Comparison for equality and inequality, denoted == and != respectively. These operations produce a Boolean result.
- Numeric comparisons, denoted by <, <=, >, and >=. These operations produce a Boolean result.
- Multiplication, denoted by *. Result has the same width as the operands. P4 architectures may impose additional restrictions—e.g., they may only allow multiplication by a power of two.
- Arithmetic shift left and right denoted by << and >>. The left operand is signed and the right operand must be either an unsigned number of type `bit<S>` or a non-negative integer literal. The

result has the same type as the left operand. Shifting by an amount greater than the width of the input produces a result where all bits are zero.

8.6.1. A note about shifts

Shifts (on signed and unsigned values) deserve a special discussion for the following reasons:

- Right shift behaves differently for signed and unsigned values: right shift for signed values is an arithmetic shift.
- Shifting with a negative amount does not have a clear semantics: the P4 type system makes it illegal to shift with a negative amount.
- Unlike C, shifting by an amount larger or equal to the number of bits has a well-defined result.
- Finally, depending on the capabilities of the target, shifting may require doing work which is exponential in the number of bits of the right-hand-side operand.

Consider the following examples:

```
bit<8> x;
bit<16> y;
... y << x ...
... y << 1024 ...
```

As mentioned above, P4 gives a precise meaning shifting with an amount larger than the size of the shifted value, unlike C.

P4 targets may impose additional restrictions on shift operations such as forbidding shifts by non-constant expressions, or by expressions whose width exceeds a certain bound. For example, a target may forbid shifting an 8-bit value by a non-constant value whose width is greater than 3 bits.

8.7. Operations on arbitrary-precision integers

The type `int` denotes arbitrary-precision integers. In P4, all expressions of type `int` must be compile-time known values. The type `int` supports the following operations:

- Negation, denoted by unary -
- Unary plus, denoted by +. This operation behaves like a no-op.
- Addition, denoted by +.
- Subtraction, denoted by -.
- Comparison for equality and inequality, denoted by == and != respectively. These operations produce a Boolean result.
- Numeric comparisons <, <=, >, and >=. These operations produce a Boolean result.
- Multiplication, denoted by *.
- Truncating integer division between positive values, denoted by /.
- Modulo between positive values, denoted by %.
- Arithmetic shift left and right denoted by << and >>. These operations produce an `int` result. The right operand must be positive. The expression `a << b` is equal to $a \times 2^b$ while `a >> b` is equal to $\lfloor a/2^b \rfloor$.

Each operand that participates in any of these operation must have type `int`. With the exception of shift, binary operations cannot be used to combine values of type `int` with values of a fixed-width type.

However, the compiler automatically inserts casts from `int` to fixed-width types in certain situations—see Section 8.9.

All computations on `int` values are carried out without loss of information. For example, multiplying two 1024-bit values may produce a 2048-bit value (note that concrete representation of `int` values is not specified). `int` values can be cast to `bit<w>` and `int<w>` values. Casting an `int` value to a fixed-width type will preserve the least-significant bits. If truncation causes significant bits to be lost, the compiler should emit a warning.

Note: bitwise-operations (`|,&,^,~`) are not defined on expressions of type `int`. In addition, it is illegal to apply division and modulo to negative values.

8.8. Operations on variable-size bit types

To support parsing headers with variable-length fields, P4 offers a type `varbit`. Each occurrence of the type `varbit` has a statically-declared maximum width, as well as a dynamic width, which must not exceed the static bound. Prior to initialization a variable-size bit-string has an unknown dynamic width.

Variable-length bit-strings support a limited set of operations:

- Parser extraction into a variable-sized bit-string using the two-argument `extract` method of a `packet_in` extern object (see Section 11.8.3). This operation sets the dynamic width of the field.
- Assignment to another variable-sized bit-string. The target of the assignment must have the same static width as the source. When executed, the assignment sets the dynamic width of the target to the dynamic width of the source.
- The `emit` method of a `packet_out` extern object, which inserts a variable-sized bit-string with a known dynamic width into the packet being constructed (see Section 14).

8.9. Casts

P4 provides a limited set of casts between types. A cast is written `(t) e`, where `t` is a type and `e` is an expression. Casts are only permitted between base types. While this design is arguably more onerous for programmers, it has several benefits:

- It makes user intent unambiguous.
- It makes the costs associated with converting numeric values explicit. Implementing certain casts involve sign-extensions, and thus can require significant computational resources on some targets.
- It reduces the number of cases that have to be considered in the P4 specification. Some targets may not support all casts.

8.9.1. Explicit casts

The following casts are legal in P4:

- `bit<1> <-> bool`: converts the value `0` to `false`, the value `1` to `true`, and vice versa.
- `int<W> -> bit<W>`: preserves all bits unchanged and reinterprets negative values as positive values
- `bit<W> -> int<W>`: preserves all bits unchanged and reinterprets values whose most-significant bit is `1` as negative values
- `bit<W> -> bit<X>`: truncates the value if `W > X`, and otherwise (i.e., if `W <= X`) pads the value with zero bits.

- `int<W> -> int<X>`: truncates the value if `W > X`, and otherwise (i.e., if `W < X`) extends it with the sign bit.
- `int -> bit<W>`: converts the integer value into a sufficiently large two's complement bit string to avoid information loss, and then truncates the result to `W` bits. The compiler should emit a warning on overflow or on conversion of negative value.
- `int -> int<W>`: converts the integer value into a sufficiently-large two's complement bit string to avoid information loss, and then truncates the result to `W` bits. The compiler should emit a warning on overflow.
- casts between two types that are introduced by `typedef` and are equivalent to one of the above combinations.

8.9.2. Implicit casts

To keep the language simple and avoid introducing hidden costs, P4 only implicitly casts from `int` to fixed-width types. In particular, applying a binary operation to an expression of type `int` and an expression with a fixed-width type will implicitly cast the `int` expression to the type of the other expression.

For example, given the following declarations,

```
bit<8>  x;
bit<16> y;
int<8>  z;
```

the compiler will add implicit casts as follows:

- `x + 1` becomes `x + (bit<8>)1`
- `z < 0` becomes `z < (int<8>)0`
- `x << 13` becomes `0`; overflow warning
- `x | 0xFFF` becomes `x | (bit<8>)0xFFF`; overflow warning

8.9.3. Illegal arithmetic expressions

Many arithmetic expressions that would be allowed in other languages are illegal in P4. To illustrate, consider the following declarations:

```
bit<8>  x;
bit<16> y;
int<8>  z;
```

The table below shows several expressions which are illegal because they do not obey the P4 typing rules. For each expression we provide several ways that the expression could be manually rewritten into a legal expression. Note that for some expression there are several legal alternatives, which may produce different results! The compiler cannot guess the user intent, so P4 requires the user to disambiguate.

Expression	Why it is illegal	Alternatives
`x + y`	Different widths	`(bit<16>)x + y` `x + (bit<8>)y`
`x + z`	Different signs	`(int<8>)x + z` `x + (bit<8>)z`
`(int<8>)y`	Cannot change both sign and width	`(int<8>)(bit<8>)y` `(int<8>)(int<16>)y`

`y + z`	Different widths and signs	`(int<8>)(bit<8>)y + z`
		`y + (bit<16>)(bit<8>)z`
		`(bit<8>)y + (bit<8>)z`
		`(int<16>)y + (int<16>)z`

`x << z`	RHS of shift cannot be signed	`x << (bit<8>)z`
`x < z`	Different signs	`X < (bit<8>)z` `(int<8>)x < z`
`1 << x`	Width of `1` is unknown	`32w1 << x`
`~1`	Bitwise operation on int	`~32w1`
`5 & -3`	Bitwise operation on int	`32w5 & -3`

8.10. Operations on tuples expressions

Tuples can be assigned to other tuples with the same type, passed as arguments and returned from functions, and can be initialized with list expressions.

```
tuple<bit<32>, bool> x = { 10, false };
```

8.11. Operations on lists

A list expression is written using curly braces, with each element separated by a comma:

```
expression ...
    | '{' expressionList '}'

expressionList
    : /* empty */
    | expression
    | expressionList ',' expression
    ;
```

The type of a list expression is a tuple type (Section 7.2.8). List expressions can be assigned to expressions of type `tuple`, `struct` or `header`, and can also be passed as arguments to methods. Lists may be nested. However, list expressions are not l-values.

For example, the following program fragment uses a list expression to pass several header fields simultaneously to a learning provider:

```
extern LearningProvider {
    void learn<T>(in T data);
```

```
}
LearningProvider() lp;

lp.learn( { hdr.ethernet.srcAddr, hdr.ipv4.src } );
```

A list may be used to initialize a structure if the list has the same number of elements as fields in the structure. The effect of such an initializer is to assign to the ith element of the list to the ith field in the structure:

```
struct S {
     bit<32> a;
     bit<32> b;
}
const S x = { 10, 20 }; //a = 10, b = 20
```

List expressions can also be used to initialize variables whose type is a `tuple` type.

```
tuple<bit<32>, bool> x = { 10, false };
```

8.12. Operations on sets

Some P4 expressions denote sets of values (`set<T>`, for some type `T`; see Section 7.2.8.1). These expressions can appear only in a few contexts—parsers and constant table entries. For example, the `select` expression (Section 11.6) has the following structure:

```
select (expression) {
     set1: state1;
     set2: state2;
    ...
}
```

Here the expressions `set1`, `set2`, etc. evaluate to sets of values and the `select` expression tests whether `expression` belongs to the sets used as labels.

```
keysetExpression
     : tupleKeysetExpression
     | simpleKeysetExpression
     ;

tupleKeysetExpression
     : '(' simpleKeysetExpression ',' simpleExpressionList ')'
     ;

simpleExpressionList
     : simpleKeysetExpression
     | simpleExpressionList ',' simpleKeysetExpression
```

```
    ;

simpleKeysetExpression
    : expression
    | DEFAULT
    | DONTCARE
    | expression MASK expression
    | expression RANGE expression
    ;
```

The mask (`&&&`) and range (`..`) operators have the same precedence, which is just higher than `&`.

8.12.1. Singleton sets

In a set context, expressions denote singleton sets. For example, in the following program fragment,

```
select (hdr.ipv4.version) {
    4: continue;
}
```

The label `4` is denotes the singleton set containing `4`.

8.12.2. The universal set

In a set context, the expressions `default` or `_` denote the universal set, which contains all possible values of a given type:

```
select (hdr.ipv4.version) {
    4: continue;
    _: reject;
}
```

8.12.3. Masks

The infix operator `&&&` takes two arguments of type `bit<W>`, and creates a value of type `set<bit<W>>`. The right value is used as a "mask", where each bit set to `0` in the mask indicates a "don't care" bit. More formally, the set denoted by `a &&& b` is defined as follows:

```
a &&& b = { c of type bit<W> where a & b = c & b }
```

For example:

```
8w0x0A &&& 8w0x0F
```

denotes a set that contains 16 different 8-bit values, whose bit-pattern is `XXXX1010`, where the value of an `X` can be any bit. Note that there may be multiple ways to express a keyset using a mask operator—e.g., `8w0xFA &&& 8w0x0F` denotes the same keyset as in the example above.

P4 architectures may impose additional restrictions on the expressions on the left and right-hand side of a mask operator: for example, they may require that either or both sub-expressions be compile-time known values.

8.12.4. Ranges

The infix operator `..` takes two arguments of the same type `T`, where `T` is either `bit<W>` or `int<W>`, and creates a value of type `set<T>`. The set contains all values numerically between the first and the second, inclusively. For example:

```
4w5 .. 4w8
```

denotes a set with values `4w5`, `4w6`, `4w7`, and `4w8`.

8.12.5. Products

Multiple sets can be combined using Cartesian product:

```
select(hdr.ipv4.ihl, hdr.ipv4.protocol) {
      (4w0x5, 8w0x1): parse_icmp;
      (4w0x5, 8w0x6): parse_tcp;
      (4w0x5, 8w0x11): parse_udp;
      (_, _): accept; }
```

The type of a product of sets is a set of tuples.

8.13. Operations on `struct` types

The only operation defined on expressions whose type is a `struct` is field access, written using dot (".") notation—e.g., `s.field`. If `s` is an l-value, then `s.field` is also an l-value. P4 also allows copying `struct`s using assignment when the source and target of the assignment have the same type. Finally, `struct`s can be initialized with a list expression, as discussed in Section 8.11.

8.14. Operations on headers

Headers provide the same operations as `struct`s. Assignment between headers also copies the "validity" header bit.

In addition, headers support the following methods:

- The method `isValid()` returns the value of the "validity" bit of the header.
- The method `setValid()` sets the header's validity bit to "true". It can only be applied to an l-value.
- The method `setInvalid()` sets the header's validity bit to "false". It can only be applied to an l-value.

The result of reading or writing a field in an invalid header is undefined. The result of reading an uninitialized header field is undefined, even if the header itself is valid.

A header object can be initialized with a list expression, similar to a `struct`—the list fields are assigned to the header fields in the order they appear. In this case the header automatically becomes valid:

```
header H { bit<32> x; bit<32> y; }
H h;
h = { 10, 12 };  // This also makes the header h valid
```

8.15. Operations on header stacks

A header stack is a fixed-size array of headers with the same type. The valid elements of a header stack need not be contiguous. P4 provides a set of computations for manipulating header stacks. A header stack `hs` of type `h[n]` can be understood in terms of the following pseudocode:

```
// type declaration
struct hs_t {
  bit<32> nextIndex;
  bit<32> size;
  h[n] data;  // Ordinary array
}

// instance declaration and initialization
hs_t hs;
hs.nextIndex = 0;
hs.size = n;
```

Intuitively, a header stack can be thought of as a struct containing an ordinary array of headers `hs` and a counter `nextIndex` that can be used to simplify the construction of parsers for header stacks, as discussed below. The `nextIndex` counter is initialized to `0`.

Given a header stack value `hs` of size `n`, the following expressions are legal:

- `hs[index]`: produces a reference to the header at the specified position within the stack; if `hs` is an l-value, the result is also an l-value. The header may be invalid. Some architectures may impose the constraint that the index expression evaluates to a compile-time known value. Accessing a header stack `hs` with an index less than `0` or greater than `hs.size` results in an undefined value.
- `hs.size`: produces a 32-bit unsigned integer that returns the size of the header stack (a compile-time constant).
- assignment from a header stack `hs` into another stack requires the stacks to have the same types and sizes. All components of `hs` are copied, including its elements and their validity bits, as well as `nextIndex`.

To help programmers write parsers for header stacks, P4 also offers computations that automatically advance through the stack as elements are parsed:

- `hs.next`: produces a reference to the element with index `hs.nextIndex` in the stack. May only be used in a `parser`. If the stack's `nextIndex` counter is greater than or equal to `size`, then evaluating this expression results in a transition to `reject` and sets the error to `error.StackOutOfBounds`. If `hs` is an l-value, then `hs.next` is also an l-value.

- `hs.last`: produces a reference to the element with index `hs.nextIndex - 1` in the stack, if such an element exists. May only be used in a `parser`. If the `nextIndex` counter is less than `1`, or greater than `size`, then evaluating this expression results in a transition to `reject` and sets the error to `error.StackOutOfBounds`. Unlike `hs.next`, the resulting reference is never an l-value.
- `hs.lastIndex`: produces a 32-bit unsigned integer that encodes the index `hs.nextIndex - 1`. May only be used in a `parser`. If the `nextIndex` counter is `0`, then evaluating this expression produces an undefined value.

Finally, P4 offers the following computations that can be used to manipulate the elements at the front and back of the stack:

- `hs.push_front(int count)`: shifts `hs` "right" by `count`. The first `count` elements become invalid. The last `count` elements in the stack are discarded. The `hs.nextIndex` counter is incremented by `count`. The `count` argument must be a positive integer that is a compile-time known value. The return type is `void`.
- `hs.pop_front(int count)`: shifts `hs` "left" by `count` (i.e., element with index `count` is copied in stack at index `0`). The last `count` elements become invalid. The `hs.nextIndex` counter is decremented by `count`. The `count` argument must be a positive integer that is a compile-time known value. The return type is `void`.

The following pseudocode defines the behavior of `push_front` and `pop_front`:

```
void push_front(int count) {
    for (int i = this.size-1; i >= 0; i -= 1) {
        if (i >= count) {
            this[i] = this[i-count];
        } else {
            this[i].setInvalid();
        }
    }
    this.nextIndex = this.nextIndex + count;
    if (this.nextIndex > this.size) this.nextIndex = this.size;
    // Note: this.last, this.next, and this.lastIndex adjust with this.nextIndex
}

void pop_front(int count) {
    for (int i = 0; i < this.size; i++) {
        if (i+count < this.size) {
            this[i] = this[i+count];
        } else {
            this[i].setInvalid();
        }
    }
    if (this.nextIndex >= count) {
        this.nextIndex = this.nextIndex - count;
```

```
    } else {
        this.nextIndex = 0;
    }
    // Note: this.last, this.next, and this.lastIndex adjust with this.nextIndex
}
```

8.16. Operations on header unions

A variable declared with a union type is initially invalid. For example:

```
header H1 {
  bit<8> f;
}
header H2 {
  bit<16> g;
}
header_union U {
  H1 h1;
  H2 h2;
}

U u; // u invalid
```

This also implies that each of the headers `h1` through `hn` contained in a header union are also initially invalid. Unlike headers, a union cannot be initialized. However, the validity of a header union can be updated by assigning a valid header to one of its elements:

```
U u;
H1 my_h1 = { 8w0 }; // my_h1 is valid
u.h1 = my_h1;       // u and u.h1 are both valid
```

We can also assign a list to an element of a header union,

```
U u;
u.h2 = { 16w1 };     // u and u.h2 are both valid
```

or set their validity bits directly.

```
U u;
u.h1.setValid();     // u and u.h1 are both valid
H1 my_h1 = u.h1;     // my_h1 is now valid, but contains an undefined value
```

Note that reading an uninitialized header produces an undefined value, even if the header is itself valid.

More formally, if `u` is an expression whose type is a header union `U` with fields ranged over by `hi`, then the following operations can be used to manipulate `u`:

- `u.hi.setValid()`: sets the valid bit for header `hi` to `true` and sets the valid bit for all other headers

to `false`, which implies that reading these headers will return an unspecified value.

- `u.hi.setInvalid()`: if the valid bit for any member header of `u` is `true` then sets it to `false`, which implies that reading any member header of `u` will return an unspecified value.

We can understand an assignment to a union

```
u.hi = e
```

as equivalent to

```
  u.hi.setValid();
  u.hi = e;
```

if `e` is valid and

```
  u.hi.setInvalid();
```

otherwise.

Assignments between variables of the same type of header union are permitted. The assignment `u1 = u2` copies the full state of header union `u2` to `u1`. If `u2` is valid, then there is some header `u2.hi` that is valid. The assignment behaves the same as `u1.hi = u2.hi`. If `u2` is not valid, then `u1` becomes invalid (i.e. if any header of `u1` was valid, it becomes invalid).

`u.isValid()` returns true if any member of the header union `u` is valid, otherwise it returns false. `setValid()` and `setInvalid()` methods are not defined for header unions.

Supplying an expression with a union type to `emit` simply emits the single header that is valid, if any.

The following example shows how we can use header unions to represent IPv4 and IPv6 headers uniformly:

```
header_union IP {
     IPv4 ipv4;
     IPv6 ipv6;
}

struct Parsed_packet {
    Ethernet ethernet;
    IP ip;
}

parser top(packet_in b, out Parsed_packet p) {
     state start {
         b.extract(p.ethernet);
         transition select(p.ethernet.etherType) {
              16w0x0800 : parse_ipv4;
              16w0x86DD : parse_ipv6;
         }
```

```
    }
    state parse_ipv4 {
        b.extract(p.ip.ipv4);
        transition accept;
    }
    state parse_ipv6 {
        b.extract(p.ip.ipv6);
        transition accept;
    }
}
```

As another example, we can also use unions to parse (selected) TCP options:

```
header Tcp_option_end_h {
    bit<8> kind;
}
header Tcp_option_nop_h {
    bit<8> kind;
}
header Tcp_option_ss_h {
    bit<8>  kind;
    bit<32> maxSegmentSize;
}
header Tcp_option_s_h {
    bit<8>  kind;
    bit<24> scale;
}
header Tcp_option_sack_h {
    bit<8>          kind;
    bit<8>          length;
    varbit<256>     sack;
}
header_union Tcp_option_h {
    Tcp_option_end_h  end;
    Tcp_option_nop_h  nop;
    Tcp_option_ss_h   ss;
    Tcp_option_s_h    s;
    Tcp_option_sack_h sack;
}

typedef Tcp_option_h[10] Tcp_option_stack;

struct Tcp_option_sack_top {
    bit<8> kind;
    bit<8> length;
}
```

```
parser Tcp_option_parser(packet_in b, out Tcp_option_stack vec) {
    state start {
        transition select(b.lookahead<bit<8>>()) {
            8w0x0 : parse_tcp_option_end;
            8w0x1 : parse_tcp_option_nop;
            8w0x2 : parse_tcp_option_ss;
            8w0x3 : parse_tcp_option_s;
            8w0x5 : parse_tcp_option_sack;
        }
    }
    state parse_tcp_option_end {
        b.extract(vec.next.end);
        transition accept;
    }
    state parse_tcp_option_nop {
         b.extract(vec.next.nop);
         transition start;
    }
    state parse_tcp_option_ss {
         b.extract(vec.next.ss);
         transition start;
    }
    state parse_tcp_option_s {
         b.extract(vec.next.s);
         transition start;
    }
    state parse_tcp_option_sack {
         bit<8> n = b.lookahead<Tcp_option_sack_top>().length;
         // n is the total length of the TCP SACK option in bytes.
         // The length of the varbit field 'sack' of the
         // Tcp_option_sack_h header is thus n-2 bytes.
         b.extract(vec.next.sack, (bit<32>) (8 * n - 16));
         transition start;
    }
}
```

8.17. Method invocations and function calls

Method invocations and function calls can be invoked using standard syntax.

```
expression
    : ...
    | expression '<' typeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'
```

```
argumentList
     : /* empty */
     | nonEmptyArgList
     ;

nonEmptyArgList
     : argument
     | nonEmptyArgList ',' argument
     ;

argument
     : expression
     | DONTCARE
     ;

typeArgumentList
     : typeRef
     | typeArgumentList ',' typeRef
     ;
```

Function arguments are evaluated in order, left to right, before the function invocation takes place. The calling convention is copy-in/copy-out (Section 6.7). For generic functions the type arguments can be explicitly specified in the function call. The compiler does not insert implicit casts for the arguments to methods or functions—the argument types must match the parameter types exactly.

The result returned by a function call is discarded when the function call is used as a statement.

The "don't care" identifier (_) can only be used for an `out` function/method argument, when the value of returned in that argument is ignored by subsequent computations. When used in generic functions or methods, the compiler may reject the program if it is unable to infer a type for the don't care argument.

8.18. Constructor invocations

Several P4 constructs denote resources that are allocated at compilation time:

- `extern` objects
- `parser`s
- `control` blocks
- `package`s

Allocation of such objects can be performed in two ways:

- Using constructor invocations, which are expressions that return an object of the corresponding type.
- Using instantiations, described in Section 9.3.

The syntax for a constructor invocation is similar to a function call. Constructors are evaluated entirely at compilation-time (see Section 16). In consequence, all constructor arguments must also be

expressions that can be evaluated at compilation time.

The following example shows a constructor invocation for setting the target-dependent implementation property of a table:

```
extern ActionProfile {
     ActionProfile(bit<32> size);  // constructor
}
table tbl {
     actions = { ... }
     implementation = ActionProfile(1024);  // constructor invocation
}
```

9. Constants and variable declarations

9.1. Constants

Constant values are defined with the syntax:

```
constantDeclaration
     : optAnnotations CONST typeRef name '=' initializer ';'
     ;

initializer
     : expression
     ;
```

Such a declaration introduces a constant whose value has the specified type. The following are all legal constant declarations:

```
const bit<32> COUNTER = 32w0x0;
struct Version {
     bit<32> major;
     bit<32> minor;
}
const Version version = { 32w0, 32w0 };
```

The `initializer` expression must be a compile-time known value.

9.2. Variables

Local variables are declared with an a type, a name, and an optional initializer (as well as an optional annotation):

```
variableDeclaration
     : annotations typeRef name optInitializer ';'
     | typeRef name optInitializer ';'
```

```
    ;

optInitializer
    : /* empty */
    | '=' initializer
    ;
```

Variable declarations without an initializer are uninitialized (except for header stacks, which have their `nextIndex` counter initialized to `0`, as discussed in 8.15). The language places few restrictions on the types of the variables: most P4 types that can be written explicitly can be used (e.g., base types, `struct`, `header`, header stack, `tuple`). However, it is impossible to declare variables with types that are only synthesized by the compiler (e.g., `set`). In addition, variables of type `parser`, `control`, `package`, or `extern` types must be declared using instantiations (see Section 9.3).

Reading the value of a variable that has not been initialized yields an undefined result. The compiler should attempt to detect and emit a warning in such situations.

Variables declarations can appear in the following locations within a P4 program:

- In a block statement,
- In a `parser` state,
- In an `action` body,
- In a `control` block apply block,
- In the list of local declarations in a `parser`, and
- In the list of local declarations in a `control`.

Variables have local scope, and behave like stack-allocated variables in languages such as C. The value of a variable is never preserved from one invocation of its enclosing block to the next. In particular, variables cannot be used to maintain state between different network packets.

9.3. Instantiations

Instantiations are similar to variable declarations, but are reserved for the types with constructors (`extern` objects, `control` blocks, `parser`s, and `package`s):

```
instantiation
    : typeRef '(' argumentList ')' name ';'
    | annotations typeRef '(' argumentList ')' name ';'
    ;
```

An instantiation is written as a constructor invocation followed by a name. Instantiations are always executed at compilation-time (Section 16.1). The effect is to allocate an object with the specified name, and to bind it to the result of the constructor invocation.

For example, a hypothetical bank of counter objects can be instantiated as follows:

```
// from target library
enum CounterType {
    Packets,
    Bytes,
```

```
    Both
}
extern Counter {
     Counter(bit<32> size, CounterType type);
     void increment(in bit<32> index);
}
// user program
control c(...) {
     Counter(32w1024, CounterType.Both) ctr;  // instantiation
     apply { ... }
}
```

9.3.1. Restrictions on top-level instantiations

A P4 program may not instantiate controls and parsers at the top-level package. This restriction is designed to ensure that most state resides in the architecture itself, or is local to a `parser` or `control`. For example, the following program is not valid:

```
// Program
control c(...) { ... }
c() c1;  // illegal top-level instantiation
```

because control `c1` is instantiated at the top-level. Note that top-level declarations of constants and instantiations of extern objects are permitted.

10. Statements

Every statement in P4 (except block statements) must end with a semicolon. Statements can appear in several places:

- Within `parser` states
- Within a `control` block
- Within an `action`

There are restrictions for the kinds of statements that can appear in each of these places. For example, conditionals are not supported in parsers, and `switch` statements are only supported in control blocks. We present here the most general case, for control blocks.

```
statement
     : assignmentOrMethodCallStatement
     | conditionalStatement
     | emptyStatement
     | blockStatement
     | exitStatement
     | returnStatement
     | switchStatement
```

```
    ;

assignmentOrMethodCallStatement
    : lvalue '(' argumentList ')' ';'
    | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
    | lvalue '='  expression ';'
    ;
```

In addition, parsers support a `transition` statement (Section 11.5).

10.1. Assignment statement

An assignment, written with the = sign, first evaluates its left sub-expression to an l-value, then evaluates its right sub-expression to a value, and finally copies the value into the l-value. Derived types (e.g. `structs`) are copied recursively, and all components of `header`s are copied, including "validity" bits. Assignment is not defined for `extern` values.

10.2. Empty statement

The empty statement, written ; is a no-op.

```
emptyStatement
    : ';'
    ;
```

10.3. Block statement

A block statement is denoted by curly braces. It contains a sequence of statements and declarations, which are executed sequentially. The variables, constants, and instantiations within a block statement are only visible within the block.

```
blockStatement
    : optAnnotations '{' statOrDeclList '}'
    ;

statOrDeclList
    : /* empty */
    | statOrDeclList statementOrDeclaration
    ;

statementOrDeclaration
    : variableDeclaration
    | constantDeclaration
    | statement
    | instantiation
    ;
```

10.4. Return statement

The `return` statement immediately terminates the execution of the `action` or `control` containing it. `return` statements are not allowed within parsers.

```
returnStatement
     : RETURN ';'
     ;
```

10.5. Exit statement

The `exit` statement immediately terminates the execution of all the blocks currently executing: the current `action` (if invoked within an `action`), the current `control`, and all its callers. `exit` statements are not allowed within parsers.

```
exitStatement
     : EXIT ';'
     ;
```

10.6. Conditional statement

The conditional statement uses standard syntax and semantics familiar from many programming languages. However, the condition expression in P4 is required to be a Boolean (and not an integer). Conditional statements may not be used within a `parser`.

```
conditionalStatement
     : IF '(' expression ')' statement
     | IF '(' expression ')' statement ELSE statement
     ;
```

When several `if` statements are nested, the `else` applies to the innermost `if` statement that does not have an `else` statement.

10.7. Switch statement

The `switch` statement can only be used within `control` blocks.

```
switchStatement
     : SWITCH '(' expression ')' '{' switchCases '}'
     ;

switchCases
     : /* empty */
     | switchCases switchCase
     ;
```

```
switchCase
     : switchLabel ':' blockStatement
     | switchLabel ':'  // fall-through
     ;

switchLabel
     : name
     | DEFAULT
     ;
```

The expression within the `switch` statement is restricted to be the result of a table invocation (See Section 12.2.2).

If a switch label is not followed by a block statement it falls through to the next label. However, if a block statement is present, it does not fall through. Note, that this is different from C-style `switch` statements, where a `break` is needed to prevent fall-through. It is legal to have no matching label for some actions, or no `default` label. At runtime, if no case matches, execution of the program simply continues. However, no label can appear twice in a switch statement.

```
switch (t.apply().action_run) {
    action1:            // fall-through to action2:
    action2: { ...}
    action3: { ...}  // no fall-through from action2 to action3 labels
}
```

Note that the `default` label of the `switch` statement is used to match on the kind of action executed, no matter whether there was a table hit or miss. The `default` label does not indicate that the table missed and the `default_action` was executed.

11. Packet parsing

This section describes the P4 constructs specific to parsing network packets.

11.1. Parser states

A P4 parser describes a state machine with one start state and two final states. The start state is always named `start`. The two final states are named `accept` (indicating successful parsing) and `reject` (indicating a parsing failure). The `start` state is part of the parser, while the `accept` and `reject` states are distinct from the states provided by the programmer and are logically outside of the parser. Figure 8 illustrates the general structure of a parser state machine.

11.2. Parser declarations

A parser declaration comprises a name, a list of parameters, an optional list of constructor parameters, local elements, and parser states (as well as optional annotations).

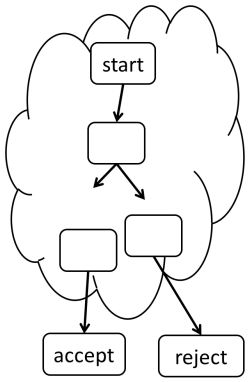

Figure 8. Parser FSM structure.

```
parserTypeDeclaration
     : optAnnotations PARSER name optTypeParameters
      '(' parameterList ')'
     ;

parserDeclaration
     : parserTypeDeclaration optConstructorParameters
       '{' parserLocalElements parserStates '}'
     ;

parserLocalElements
     : /* empty */
     | parserLocalElements parserLocalElement
     ;

parserStates
     : parserState
     | parserStates parserState
     ;
```

For a description of `optConstructorParameters`, which are useful for building parameterized parsers, see Section 13.

Unlike parser type declarations, parser declarations may not be generic—e.g., the following declaration is illegal:

```
parser P<H>(inout H data) { ... }
```

Hence, used in the context of a `parserDeclaration` the production rule `parserTypeDeclaration` should not yield type parameters.

At least one state, named `start`, must be present in any `parser`. A parser may not define two states with the same name. It is also illegal for a parser to give explicit definitions for the `accept` and `reject` states—those states are logically distinct from the states defined by the programmer.

State declarations are described below. Preceding the parser states, a `parser` may also contain a list of local elements. These can be constants, variables, or instantiations of objects that may be used within the parser. Such objects may be instantiations of `extern` objects, or other `parser`s that may be invoked as subroutines. However, it is illegal to instantiate a `control` block within a `parser`.

```
parserLocalElement
     : constantDeclaration
     | variableDeclaration
     | instantiation
     ;
```

For an example containing a complete declaration of a parser see Section 5.3.

11.3. The Parser abstract machine

The semantics of a P4 parser can be formulated in terms of an abstract machine that manipulates a `ParserModel` data structure. This section describes this abstract machine in pseudo-code.

A parser starts execution in the `start` state and ends execution when one of the `reject` or `accept` states has been reached.

```
ParserModel {
     error        parseError;
     onPacketArrival(packet p) {
          ParserModel.parseError = error.NoError;
          goto start;
     }
}
```

An architecture must specify the behavior when the `accept` and `reject` states are reached. For example, an architecture may specify that all packets reaching the `reject` state are dropped without further processing. Alternatively, it may specify that such packets are passed to the next block after the parser, with intrinsic metadata indicating that the parser reached the `reject` state, along with the error recorded.

11.4. Parser states

A parser state is declared with the following syntax:

```
parserState
     : optAnnotations STATE name
      '{' parserStatements transitionStatement '}'
     ;
```

Each state has a name and a body. The body consists of a sequence of statements that describe the processing performed when the parser transitions to that state including:

- Local variable declarations,
- Assignment statements,
- Method calls, which serve several purposes:

  - Invoking functions (e.g., using `verify` to check the validity of data already parsed), and
  - Invoking methods (e.g., extracting data out of packets or computing checksums) and other parsers (see Section 11.10), and

- Transitions to other states (discussed in Section 11.5).

The syntax for parser statements is given by the following grammar rules:

```
parserStatements
     : /* empty */
     | parserStatements parserStatement
     ;

parserStatement
     : assignmentOrMethodCallStatement
     | variableDeclaration
     | constantDeclaration
     | parserBlockStatement
     ;

parserBlockStatement
     : optAnnotations '{' parserStatements '}'
     ;
```

Architectures may place restrictions on the expressions and statements that can be used in a parser—e.g., they may forbid the use of operations such as multiplication or place restrictions on the number of local variables that may be used.

In terms of the `ParserModel`, the sequence of statements in a state are executed sequentially.

11.5. Transition statements

The last statement in a parser state is an optional `transition` statement, which transfers control to another state, possibly `accept` or `reject`. A `transition` statements is written using the following syntax:

```
transitionStatement
     : /* empty */
     | TRANSITION stateExpression
     ;

stateExpression
     : name ';'
     | selectExpression
     ;
```

The execution of the transition statement causes `stateExpression` to be evaluated, and transfers control to the resulting state.

In terms of the `ParserModel`, the semantics of a `transition` statement can be formalized as follows:

```
goto eval(stateExpression)
```

For example, this statement:

```
transition accept;
```

terminates execution of the current parser and transitions immediately to the `accept` state.

If the body of a state block does not end with a `transition` statement, the implied statement is

```
transition reject;
```

11.6. Select expressions

A `select` expression evaluates to a state. The syntax for a `select` expression is as follows:

```
selectExpression
    : SELECT '(' expressionList ')' '{' selectCaseList '}'
    ;

selectCaseList
    : /* empty */
    | selectCaseList selectCase
    ;

selectCase
    : keysetExpression ':' name ';'
    ;
```

In a `select` expression, if the `expressionList` has type `tuple<T>`, then each `keysetExpression` must have type `set<tuple<T>>`.

In terms of the `ParserModel`, the meaning of a select expression:

```
select(e) {
    ks[0]: s[0];
    ks[1]: s[1];
    ...
    ks[n-2]: s[n-1];
    _ : sd;  // ks[n-1] is default
}
```

is defined in pseudo-code as:

```
key = eval(e);
for (int i=0; i < n; i++) {
    keyset = eval(ks[i]);
    if (keyset.contains(key)) return s[i];
```

```
}
verify(false, error.NoMatch);
```

Some targets may require that all keyset expressions in a select expression be compile-time known values. Keysets are evaluated in order, from top to bottom as implied by the pseudo-code above; the first keyset that includes the value in the `select` argument provides the result state. If no label matches, the execution triggers a runtime error with the standard error code `error.NoMatch`.

Note that this implies that all cases after a `default` or _ label are unreachable; the compiler should emit a warning if it detects unreachable cases. This constitutes an important difference between `select` expressions and the `switch` statements found in many programming languages since the keysets of a `select` expression may "overlap".

The typical way to use a `select` expression is to compare the value of a recently-extracted header field against a set of constant values, as in the following example:

```
header IPv4_h { ... bit<8> protocol; ... }
struct P { ... IPv4_h ipv4; ... }
P headers;
select (headers.ipv4.protocol) {
     8w6   : parse_tcp;
     8w17  : parse_udp;
     _     : accept;
}
```

For example, to detect TCP reserved ports (< 1024) one could write:

```
select (p.tcp.port) {
     16w0 &&& 16w0xFC00: well_known_port;
     _: other_port;
}
```

The expression `16w0 &&& 16w0xFC00` describes the set of 16-bit values whose most significant six bits are zero.

11.7. verify

The `verify` statement provides a simple form of error handling. `verify` can only be invoked within a parser; it is used syntactically as if it were a function with the following signature:

```
extern void verify(in bool condition, in error err);
```

If the first argument is `true`, then executing the statement has no side-effect. However, if the first argument is `false`, it causes an immediate transition to `reject`, which causes immediate parsing termination; at the same time, the `parserError` associated with the parser is set to the value of the second argument.

In terms of the `ParserModel` the semantics of a `verify` statement is given by:

```
ParserModel.verify(bool condition, error err) {
     if (condition == false) {
          ParserModel.parserError = err;
          goto reject;
     }
}
```

11.8. Data extraction

The P4 core library contains the following declaration of a built-in `extern` type called `packet_in` that represents incoming network packets. The `packet_in` extern is special: it cannot be instantiated by the user explicitly. Instead, the architecture supplies a separate instance for each `packet_in` argument to a `parser` instantiation.

```
extern packet_in {
     void extract<T>(out T headerLvalue);
     void extract<T>(out T variableSizeHeader, in bit<32> varFieldSizeBits);
     T lookahead<T>();
     bit<32> length();  // This method may be unavailable in some architectures
     void advance(bit<32> bits);
}
```

To extract data from a packet represented by an argument `b` with type `packet_in`, a parser invokes the `extract` methods of `b`. There are two variants of the `extract` method: a one-argument variant for extracting fixed-size headers, and a two-argument variant for extracting variable-sized headers. Because these operations can cause runtime verification failures (see below), these methods can only be executed within parsers.

When extracting data into a bit-string or integer, the first packet bit is extracted to the most significant bit of the integer.

Some targets may perform cut-through packet processing, i.e., they may start processing a packet before its length is known (i.e., before all bytes have been received). On such a target calls to the `packet_in.length()` method cannot be implemented. Attempts to call this method should be flagged as errors (either at compilation time by the compiler back-end, or when attempting to load the compiled P4 program onto a target that does not support this method).

In terms of the `ParserModel`, the semantics of `packet_in` can be captured using the following abstract model of packets:

```
packet_in {
     unsigned nextBitIndex;
     byte[] data;
     unsigned lengthInBits;
     void initialize(byte[] data) {
          this.data = data;
          this.nextBitIndex = 0;
          this.lengthInBits = data.sizeInBytes * 8;
```

```
    }
    bit<32> length() { return this.lengthInBits / 8; }
}
```

11.8.1. Fixed width extraction

The single-argument `extract` method handles fixed-width headers, and is declared in P4 as follows:

```
void extract<T>(out T headerLeftValue);
```

The expression `headerLeftValue` must evaluate to a l-value (see Section 6.6) of type `header` with a fixed width. If this method executes successfully, on completion the `headerLvalue` is filled with data from the packet and its validity bit is set to `true`. This method may fail in various ways—e.g., if there are not enough bits left in the packet to fill the specified header.

For example, the following program fragment extracts an Ethernet header:

```
struct Result { ... Ethernet_h ethernet; ... }
parser P(packet_in b, out Result r) {
    state start {
        b.extract(r.ethernet);
    }
}
```

In terms of the `ParserModel`, the semantics of the single-argument `extract` is given in terms of the following pseudo-code method, using data from the `packet` class defined above. We use the special `valid$` identifier to indicate the hidden valid bit of a header, `isNext$` to indicate that the l-value was obtained using `next`, and `nextIndex$` to indicate the corresponding header stack properties.

```
void packet_in.extract<T>(out T headerLValue) {
   bitsToExtract = sizeofInBits(headerLValue);
   lastBitNeeded = this.nextBitIndex + bitsToExtract;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
   headerLValue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
   headerLValue.valid$ = true;
   if headerLValue.isNext$ {
     verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
     headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
   }
   this.nextBitIndex += bitsToExtract;
}
```

11.8.2. Variable width extraction

The two-argument `extract` handles variable-width headers, and is declared in P4 as follows:

```
void extract<T>(out T headerLvalue, in bit<32> variableFieldSize);
```

The expression `headerLvalue` must be a l-value representing a header that contains exactly one `varbit` field. The expression `variableFieldSize` must evaluate to a `bit<32>` value that indicates the number of bits to be extracted into the unique `varbit` field of the header (i.e., this size is not the size of the complete header, just the `varbit` field).

In terms of the `ParserModel`, the semantics of the two-argument `extract` is captured by the following pseudo-code:

```
void packet_in.extract<T>(out T headerLvalue,
                          in bit<32> variableFieldSize) {
    bitsToExtract = sizeOfFixedPart(headerLvalue) + variableFieldSize;
    lastBitNeeded = this.nextBitIndex + bitsToExtract;
    ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
    ParserModel.verify(bitsToExtract <= headerLvalue.maxSize, error.HeaderTooShort);
    headerLvalue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
    headerLvalue.varbitField.size = variableFieldSize;
    headerLvalue.valid$ = true;
    if headerLValue.isNext$ {
      verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
      headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
    }
    this.nextBitIndex += bitsToExtract;
}
```

The following example shows one way to parse IPv4 options—by splitting the IPv4 header into two separate headers:

```
// IPv4 header without options
header IPv4_no_options_h {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    bit<32>   srcAddr;
    bit<32>   dstAddr;
}
header IPv4_options_h {
    varbit<320> options;
}
```

```
struct Parsed_headers {
     ...
     IPv4_no_options_h ipv4;
     IPv4_options_h     ipv4options;
}

error { InvalidIPv4Header }

parser Top(packet_in b, out Parsed_headers headers) {
    ...
    state parse_ipv4 {
         b.extract(headers.ipv4);
         verify(headers.ipv4.ihl >= 5, error.InvalidIPv4Header);
         transition select (headers.ipv4.ihl) {
              5: dispatch_on_protocol;
              _: parse_ipv4_options;
    }

    state parse_ipv4_options {
         // use information in the ipv4 header to compute the number
         // of bits to extract
         b.extract(headers.ipv4options,
                    (bit<32>)(((bit<16>)headers.ipv4.ihl - 5) * 32));
         transition dispatch_on_protocol;
    }
}
```

11.8.3. Lookahead

The `lookahead` method provided by the `packet_in` packet abstraction evaluates to a set of bits from the input packet without advancing the `nextBitIndex` pointer. Similar to `extract`, it will transition to `reject` and set the error if there are not enough bits in the packet. The `lookahead` method can be invoked as follows,

```
b.lookahead<T>()
```

where `T` must be a type with fixed width. In case of success the result of the evaluation of `lookahead` returns a value of type `T`.

In terms of the `ParserModel`, the semantics of `lookahead` is given by the following pseudo-code:

```
T packet_in.lookahead<T>() {
    bitsToExtract = sizeof(T);
    lastBitNeeded = this.nextBitIndex + bitsToExtract;
    ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
    T tmp = this.data.extractBits(this.nextBitIndex, bitsToExtract);
```

```
    return tmp;
}
```

The TCP options example from Section 8.16 also illustrates how `lookahead` can be used:

```
state start {
     transition select(b.lookahead<bit<8>>()) {
          0: parse_tcp_option_end;
          1: parse_tcp_option_nop;
          2: parse_tcp_option_ss;
          3: parse_tcp_option_s;
          5: parse_tcp_option_sack;
     }
}
...
state parse_tcp_option_sack {
     bit<8> n = b.lookahead<Tcp_option_sack_top>().length;
     b.extract(vec.next.sack, (bit<32>) (8 * n - 16));
     transition start;
}
```

11.8.4. Skipping bits

P4 provides two ways to skip over bits in an input packet without assigning them to a header:

One way is to `extract` to the underscore identifier, explicitly specifying the type of the data:

```
b.extract<T>(_)
```

Another way is to use the `advance` method of the packet when the number of bits to skip is known.

In terms of the `ParserModel`, the meaning of `advance` is given in pseudo-code as follows:

```
void packet_in.advance(bit<32> bits) {
    lastBitNeeded = this.nextBitIndex + bits;
    ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
    this.nextBitIndex += bits;
}
```

11.9. Header stacks

A header stack has two properties, `next` and `last`, which can be used in parsing. Consider the following declaration, which defines a stack for representing the headers of a packet with at most ten MPLS headers:

```
header Mpls_h {
     bit<20> label;
```

```
    bit<3>  tc;
    bit     bos;
    bit<8>  ttl;
}
Mpls_h[10] mpls;
```

The expression `mpls.next` represents an l-value of type `Mpls_h` that references an element in the `mpls` stack. Initially, `mpls.next` refers to the first element of stack. It is automatically advanced on each successful call to `extract`. The `mpls.last` property refers to the element immediately preceding `next` if such an element exists. Attempting to access `mpls.next` element when the stack's `nextIndex` counter is greater than or equal to `size` causes a transition to `reject` and sets the error to `error.StackOutOfBounds`. Likewise, attempting to access `mpls.last` when the `nextIndex` counter is equal to `0` causes a transition to `reject` and sets the error to `error.StackOutOfBounds`.

The following example shows a simplified parser for MPLS processing:

```
struct Pkthdr {
    Ethernet_h ethernet;
    Mpls_h[3] mpls;
    // other headers omitted
}

parser P(packet_in b, out Pkthdr p) {
    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x8847: parse_mpls;
            0x0800: parse_ipv4;
        }
    }
    state parse_mpls {
        b.extract(p.mpls.next);
        transition select(p.mpls.last.bos) {
            0: parse_mpls; // This creates a loop
            1: parse_ipv4;
        }
    }
    // other states omitted
}
```

11.10. Sub-parsers

P4 allows parsers to invoke the services of other parsers, similar to subroutines. To invoke the services of another parser, the sub-parser must be first instantiated; the services of an instance are invoked by calling it using its apply method.

The following example shows a sub-parser invocation:

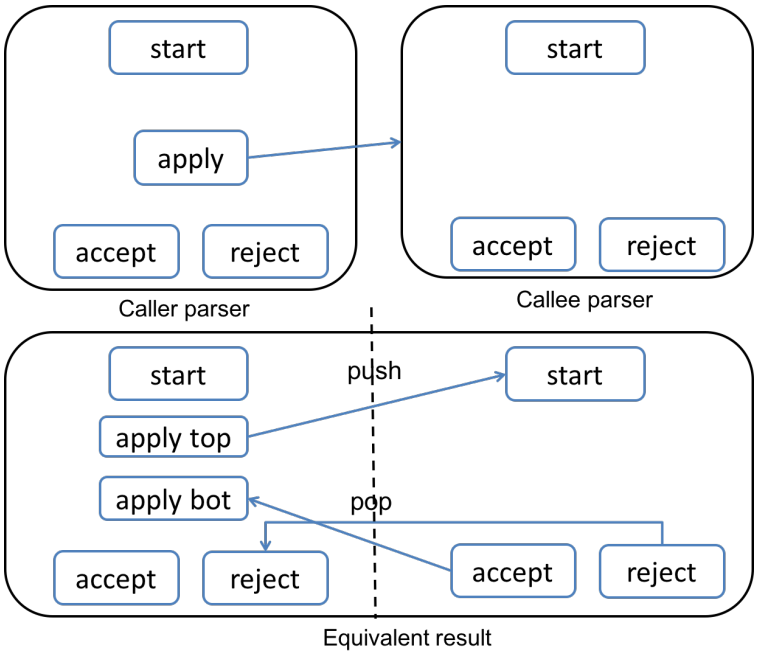

Figure 9. Semantics of invoking a sub-parser: top: original program, bottom: equivalent program.

```
parser callee(packet_in packet, out IPv4 ipv4) { ...}
parser caller(packet_in packet, out Headers h) {
      callee() subparser;  // instance of callee
      state subroutine {
            subparser.apply(packet, h.ipv4);  // invoke sub-parser
      }
}
```

The semantics of a sub-parser invocation can be described as follows:

- The state invoking the sub-parser is split into two half-states at the parser invocation statement.
- The top half includes a transition to the sub-parser `start` state.
- The sub-parser's `accept` state is identified with the bottom half of the current state
- The sub-parser's `reject` state is identified with the reject state of the current parser.

Figure 9 shows a diagram of this process.

Note that since P4 requires declarations to precede uses, it is impossible to create recursive (or mutually recursive) parsers.

Architectures may impose (static or dynamic) constraints on the number of parser states that can be traversed for processing each packet. For example, a compiler for a specific target may reject parsers containing loops that cannot be unrolled at compilation time or that may contain cycles that do not advance the cursor. If a parser aborts execution dynamically because it exceeded the time budget allocated for parsing, the parser should transition to `reject` and set the standard error `error.ParserTimeout`.

12. Control blocks

P4 parsers are responsible for extracting bits from a packet into headers. These headers (and other metadata) can be manipulated and transformed within `control` blocks. The body of a control block resembles a traditional imperative program. Within the body of a control block, match-action units can be invoked to perform data transformations. Match-action units are represented in P4 by constructs called `tables`.

Syntactically, a `control` block is declared with a name, parameters, optional type parameters, and a sequence of declarations of constants, variables, `action`s, `table`s, and other instantiations:

```
controlDeclaration
    : controlTypeDeclaration optConstructorParameters
      /* controlTypeDeclaration cannot contain type parameters */
      '{' controlLocalDeclarations APPLY controlBody '}'
    ;

controlLocalDeclarations
    : /* empty */
    | controlLocalDeclarations controlLocalDeclaration
    ;

controlLocalDeclaration
    : constantDeclaration
    | variableDeclaration
    | actionDeclaration
    | tableDeclaration
    | instantiation
    ;

controlBody
    : blockStatement
    ;
```

It is illegal to instantiate a `parser` within a `control` block. For a description of the `optConstructorParameters`, which can be used to build parameterized control blocks, see Section 13.

Unlike control type declarations, control declarations may not be generic—e.g., the following declaration is illegal:

```
control C<H>(inout H data) { ... }
```

P4 does not support exceptional control-flow within a `control` block. The only statement which has a non-local effect on control flow is `exit`, which causes execution of the enclosing control block to immediately terminate. That is, there is no equivalent of the `verify` statement or the `reject` state from parsers. Hence, all error handling must be performed explicitly by the programmer.

The rest of this section describes the core components of a `control` block, starting with actions.

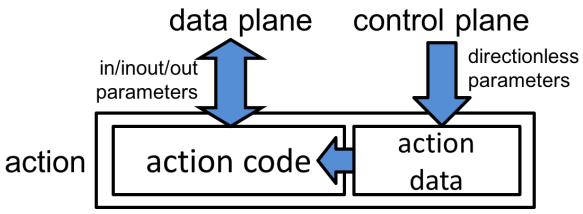

Figure 10. Actions contain code and data. The code is in the P4 program, while the data is set by the control plane. Parameters are bound by the data plane.

12.1. Actions

Actions are code fragments that can read and write the data being processed. Actions may contain data values that can be written by the control plane and read by the data plane. Actions are the main construct by which the control-plane can influence dynamically the behavior of the data plane. Figure 10 shows the abstract model of an `action`.

```
actionDeclaration
    : optAnnotations ACTION name '(' parameterList ')' blockStatement
    ;
```

Syntactically actions resemble functions with no return value. Actions may be declared within a control block; in this case they can only be used within instances of that control block.

The following example shows an action declaration:

```
action Forward_a(out bit<9> outputPort, bit<9> port) {
    outputPort = port;
}
```

Action parameters may not have `extern` types. Action parameters that have no direction (e.g., `port` in the previous example) indicate "action data." All such parameters must appear at the end of the parameter list. When used in a match-action table (see Section 12.2.1.2), these parameters will be provided by the control plane.

The body of an action consists of a sequence of statements and declarations. No `switch` statements are allowed within an action—the grammar permits them, but a semantic check should reject them. Some targets may impose additional restrictions on action bodies—e.g., only allowing straight-line code, with no conditional statements or expressions.

12.1.1. Invoking actions

Actions can be executed in two ways:

- Implicitly: by tables during match-action processing.
- Explicitly: either from a `control` block or from another `action`. In either case, the values for all action parameters must be supplied explicitly, including values for the directionless parameters. In this case, the directionless parameters behave like `in` parameters.

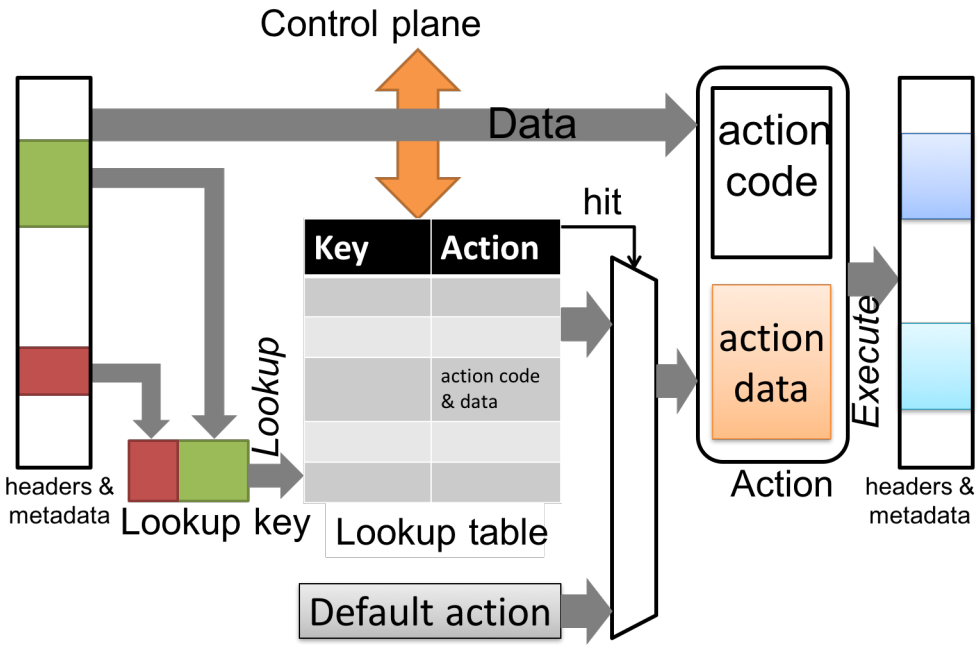

Figure 11. Match-Action Unit Dataflow.

12.2. Tables

A `table` describes a match-action unit. The structure of a match-action unit is shown in Figure 11. Processing a packet using a match-action table executes the following steps:

- Key construction.
- Key lookup in a lookup table (the "match" step). The result of key lookup is an "action".
- Action execution (the "action step") over the input data, resulting in mutations of the data.

A `table` declaration introduces a table instance. To obtain multiple instances of a table, it must be declared within a control block that is itself instantiated multiple times.

The look-up table is a finite map whose contents are manipulated asynchronously (read/write) by the target control-plane, through a separate control-plane API (see Figure 11). Note that the term "table" is overloaded: it can refer to the P4 `table` objects that appear in P4 programs, as well as the internal look-up tables used in targets. We will use the term "match-action unit" when necessary to disambiguate.

Syntactically a table is defined in terms of a set of key-value properties. Some of these properties are "standard" properties, but the set of properties can be extended by target-specific compilers as needed.

```
tableDeclaration
    : optAnnotations TABLE name '{' tablePropertyList '}'
    ;

tablePropertyList
```

```
    : tableProperty
    | tablePropertyList tableProperty
    ;

tableProperty
    : KEY '=' '{' keyElementList '}'
    | ACTIONS '=' '{' actionList '}'
    | CONST ENTRIES '=' '{' entriesList '}' /* immutable entries */
    | optAnnotations CONST IDENTIFIER '=' initializer ';'
    | optAnnotations IDENTIFIER '=' initializer ';'
    ;
```

The standard table properties include:

- `key`: An expression that describes how the key used for look-up is computed.
- `actions`: A list of all actions that may be found in the table.

In addition, the tables may optionally define the following property,

- `default_action`: an action to execute when the lookup in the lookup table fails to find a match for the key used.

The compiler may set the `default_action` to `NoAction` (and also insert it into the list of `actions`) for tables that do not define the `default_action` property. This is consistent with the semantics given in Section 12.2.1.3. In this document, we assume that that this transformation has been performed, so that all tables have a `default_action` property.

In addition, tables may contain architecture-specific properties (see Section 12.2.1.5).

A property marked as `const` cannot be changed dynamically by the control-plane. The `key` and `actions` properties are always constant, so the `const` keyword is not needed for these.

12.2.1. Table properties

12.2.1.1. Keys The `key` is a table property which specifies the data plane values that should be used to look up an entry. A key is a list of pairs of the form `(e : m)`, where `e` is an expression that describes the data to be matched in the table, and `m` is a `match_kind` constant that describes the algorithm used to perform the lookup (see Section 7.1.3).

```
keyElementList
    : /* empty */
    | keyElementList keyElement
    ;

keyElement
    : expression ':' name optAnnotations ';'
    ;
```

For example, consider the following program fragment:

```
table Fwd {
     key = {
         ipv4header.dstAddress : ternary;
         ipv4header.version    : exact;
     }
     ...
}
```

Here the key comprises two fields from the `ipv4header` header: `dstAddress` and `version`. The `match_kind` constants serve three purposes:

- They specify the algorithm used to match data plane values against the entries in the table at runtime.
- They are used to synthesize the control-plane API that is used to populate the table.
- They are used by the compiler back-end to allocate resources for the implementation of the table.

The P4 core library contains three predefined `match_kind` identifiers:

```
match_kind {
    exact,
    ternary,
    lpm
}
```

These identifiers correspond to the $P4_{14}$ match kinds with the same names. The semantics of these annotations is actually not needed to describe the behavior of the P4 abstract machine; how they are used influences only the control-plane API and the implementation of the look-up table. From the point of view of the P4 program, a look-up table is an abstract finite map that is given a key and produces as a result either an action or a "miss" indication, as described in Section 12.2.3.

If a table has no `key` property, then it contains no look-up table, just a default action—i.e., the associated lookup table is always the empty map.

Each key element can have an optional `@name` annotation which is used to synthesize the control-plane visible name for the key field.

12.2.1.2. Actions A table must declare all possible actions that may appear within the associated lookup table or in the default action. This is done with the `actions` property; the value of this property is always an `actionList`:

```
actionList
     : /* empty */
     | actionList actionRef ';'
     ;

actionRef
     : optAnnotations name
     | optAnnotations name '(' argumentList ')'
```

```
    ;
```

To illustrate, recall the example Very Simple Switch program in Section 5.3:

```
action Drop_action() {
  outCtrl.outputPort = DROP_PORT;
}

action Rewrite_smac(EthernetAddress sourceMac) {
  headers.ethernet.srcAddr = sourceMac;
}

table smac {
    key = { outCtrl.outputPort : exact; }
    actions = {
        Drop_action;
        Rewrite_smac;
    }
}
```

- The entries in the `smac` `table` may contain two different actions: `Drop_action` and `Rewrite_mac`.
- The `Rewrite_smac` action has one parameter, `sourceMac`, which is bound by the control plane.

Each action in the list of actions for a table must have a distinct name—e.g., the following program fragment is illegal:

```
action a() {}
control c() {
    action a() {}
    // Illegal table: two actions with the same name
    table t { actions = { a; .a; } }
}
```

Each action parameter that has a direction (`in`, `inout`, or `out`) must be bound in the `actions` list specification; conversely, no directionless parameters may be bound in the list. The expressions supplied as arguments to an `action` are not evaluated until the action is invoked.

```
action a(in bit<32> x) { ...}
bit<32> z;
action b(inout bit<32> x, bit<8> data) { ...}
table t {
    actions = {
       // a; -- illegal, x parameter must be bound
       a(5);  // binding a's parameter x to 5
       b(z);  // binding b's parameter x to z
       // b(z, 3); -- illegal, cannot bind directionless data parameter
```

```
        // b(); -- illegal, x parameter must be bound
    }
}
```

12.2.1.3. Default action The default action for a table is an action that is invoked automatically by the match-action unit whenever the lookup table does not find a match for the supplied key.

If present, the `default_action` property must appear after the `action` property. It may be declared as `const`, indicating that it cannot be changed dynamically by the control-plane. The `default action` must be one of the actions that appear in the actions list. In particular, the expressions passed as `in`, `out`, or `inout` parameters must be syntactically identical to the expressions used in one of the elements of the `actions` list.

For example, in the above `table` we could set the default action as follows (marking it also as constant):

```
const default_action = Rewrite_smac(48w0xAA_BB_CC_DD_EE_FF);
```

Note that the specified default action must supply arguments for the control-plane bound parameters (i.e., the directionless parameters), since the action is synthesized at compilation time. The expressions supplied as arguments for parameters with a direction (`in`, `inout`, or `out`) are evaluated when the action is invoked while the expressions supplied as arguments for directionless parameters are evaluated at compile time.

Continuing the example from the previous section, following are several legal and illegal specifications of default actions for the `table t`:

```
  default_action = a(5); // OK - no control-plane parameters
  // default_action = a(z); -- illegal, a's x parameter is already bound to 5
  default_action = b(z,8w8); // OK - bind b's data parameter to 8w8
  // default_action = b(z); -- illegal, b's data parameter is not bound
  // default_action = b(x, 3); -- illegal: x parameter of b bound to x instead of z
```

If a table does not specify the `default_action` property and no entry matches a given packet, then the table does not affect the packet and processing continues according to the imperative control flow of the program.

12.2.1.4. Entries While table entries are typically installed by the control plane, tables may also be initialized at compile-time with a set of entries. This is useful in situations where tables are used to implement fixed algorithms—defining table entries statically enables expressing these algorithm directly in P4, which allows the compiler to infer how the table is actually used and potentially make better allocation decisions for targets with limited resources. Entries declared in the P4 source are installed in the table when the program is loaded onto the target.

Table entries are defined using the following syntax:

```
tableProperty
 : const ENTRIES '=' '{' entriesLlist '}' /* immutable entries */
```

```
entriesList
 : entry
 | entriesList entry

entry
 : keysetExpression ':' actionRef optAnnotations ';'
```

Table entries are immutable (const)—i.e., they can only be read and cannot be changed or removed by the control plane. It follows that tables that define entries in the P4 source are immutable. This design choice has important ramifications for the P4 runtime since it does not have to keep track of different types of entries in one table (mutable and immutable). Future versions of P4 may add the ability to mix mutable and immutable entries in the same table, by declaring additional `entries` properties without the `const` keyword.

The `keysetExpression` component of an entry is a tuple that must provide a field for each key in the table keys (see Sec. 12.2.1). The table key type must match the type of the element of the set. `actionRef` must be an action which appears in the table actions list, with all its arguments bound.

Entries in a table are matched in the program order, stopping at the first matching entry.

Depending on the `match_kind` of the keys, key set expressions may define one or multiple entries. The compiler will synthesize the correct number of entries to be installed in the table. Target constraints may further restrict the ability of synthesizing entries. For example, if the number of synthesized entries exceeds the table size, the compiler implementation may choose to issue a warning or an error, depending on target capabilities.

To illustrate, consider the following example:

```
header hdr {
     bit<8>  e;
     bit<16> t;
     bit<8>  l;
     bit<8>  r;
     bit<1>  v;
}

struct Header_t {
     hdr h;
}
struct Meta_t {}

control ingress(inout Header_t h, inout Meta_t m,
                  inout standard_metadata_t standard_meta) {

     action a() { standard_meta.egress_spec = 0; }
     action a_with_control_params(bit<9> x) { standard_meta.egress_spec = x; }

     table t_exact_ternary {
```

```
      key = {
            h.h.e : exact;
            h.h.t : ternary;
        }

    actions = {
            a;
            a_with_control_params;
        }

    default_action = a;

    const entries = {
            (0x01, 0x1111 &&& 0xF    ) : a_with_control_params(1);
            (0x02, 0x1181             ) : a_with_control_params(2);
            (0x03, 0x1111 &&& 0xF000) : a_with_control_params(3);
            (0x04, 0x1211 &&& 0x02F0) : a_with_control_params(4);
            (0x04, 0x1311 &&& 0x02F0) : a_with_control_params(5);
            (0x06, _                  ) : a_with_control_params(6);
        }
    }

}
```

In this example we define a set of 6 entries that cause the invocation of action `a_with_control_params`. Once the program is loaded, these entries are installed in the table in the order they are enumerated in the program.

12.2.1.5. Additional properties A `table` declaration defines its essential control and data plane interfaces—i.e., keys and actions. However, the best way to implement a table may actually depend on the nature of the entries that will be installed at runtime (for example, tables could be dense or sparse, could be implemented as hash-tables, associative memories, tries, etc.) In addition, some architectures may support extra table properties whose semantics lies outside the scope of this specification. For example, in architectures where table resources are statically allocated, programmers may be required to define a `size` table property, which can be used by the compiler back-end to allocate storage resources. However, these architecture-specific properties may not change the semantics of table lookups, which always produce either a `hit` and an action or a `miss`—they can only change how those results are interpreted on the state of the data plane. This restriction is needed to ensure that it is possible to reason about the behavior of tables during compilation.

As another example, an `implementation` property could be used to pass additional information to the compiler back-end. The value of this property could be an instance of an `extern` block chosen from a suitable library of components. For example, the core functionality of the $P4_{14}$ table `action_profile` constructs could be implemented on architectures that support this feature using a construct such as the following:

```
extern ActionProfile {
    ActionProfile(bit<32> size); // number of distinct actions expected
}
table t {
     key = { ...}
     size = 1024;
     implementation = ActionProfile(32);  // constructor invocation
}
```

Here the action profile might be used to optimize for the case where the table has a large number of entries, but the actions associated with those entries are expected to range over a small number of distinct values. Introducing a layer of indirection enables sharing identical entries, which can significantly reduce the table's storage requirements.

12.2.2. Match-action unit invocation

A `table` can be invoked by calling its `apply` method. Calling an apply method on a table instance returns a value with a `struct` type with two fields. This structure is synthesized by the compiler automatically. For each `table T`, the compiler synthesizes an `enum` and a `struct`, shown in pseudo-P4:

```
enum action_list(T) {
    // one field for each action in the actions list of table T
}
struct apply_result(T) {
     bool hit;
     action_list(T) action_run;
}
```

The evaluation of the `apply` method sets the `hit` field to `true` if a match is found in the lookup-table. This bit can be used to drive the execution of the control-flow in the control block that invoked the table:

```
if (ipv4_match.apply().hit) {
     // there was a hit
} else {
     // there was a miss
}
```

The `action_run` field indicates which kind of action was executed (irrespective of whether it was a hit or a miss). It can be used in a switch statement:

```
switch (dmac.apply().action_run) {
     Drop_action: { return; }
}
```

12.2.3. Match-action unit execution semantics

The semantics of a table invocation statement:

```
m.apply();
```

is given by the following pseudo-code (see also Figure 11):

```
apply_result(m) m.apply() {
     apply_result(m) result;

     var lookupKey = m.buildKey(m.key); // using key block
     action RA = m.table.lookup(lookupKey);
     if (RA == null) {       // miss in lookup table
        result.hit = false;
        RA = m.default_action;  // use default action
     }
     else {
        result.hit = true;
     }
     result.action_run = action_type(RA);
     evaluate_and_copy_in_RA_args(RA);
     execute(RA);
     copy_out_RA_args(RA);
     return result;
}
```

12.3. The Match-Action Pipeline Abstract Machine

We can describe the computational model of a match-action pipeline, embodied by a control block: the body of the control block is executed, similarly to the execution of a traditional imperative program:

- At runtime, statements within a block are executed in the order they appear in the control block.
- Execution of the `return` statement causes immediate termination of the execution of the current `control` block, and a return to the caller.
- Execution of the `exit` statement causes the immediate termination of the execution of the current `control` block and of all the enclosing caller `control` blocks.
- Applying a `table` executes the corresponding match-action unit, as described above.

12.4. Invoking controls

P4 allows controls to invoke the services of other controls, similar to subroutines. To invoke the services of another control, it must be first instantiated; the services of an instance are invoked by calling it using its `apply` method.

The following example shows a control invocation:

```
control Callee(inout IPv4 ipv4) { ...}
control Caller(inout Headers h) {
      Callee() instance;  // instance of callee
      apply {
            instance.apply(h.ipv4);  // invoke control
      }
}
```

13. Parameterization

In order to support libraries of useful P4 components, both `parser`s and `control` blocks can be additionally parameterized through the use of constructor parameters.

Consider again the parser declaration syntax:

```
parserDeclaration
     : parserTypeDeclaration optConstructorParameters
       '{' parserLocalElements parserStates '}'
     ;

optConstructorParameters
     : /* empty */
     | '(' parameterList ')'
     ;
```

From this grammar fragment we infer that a `parser` declaration may have two sets of parameters:

- The runtime parser parameters (`parameterList`)
- Optional parser constructor parameters (`optConstructorParameters`)

Constructor parameters must be directionless (i.e., they cannot be `in`, `out`, or `inout`) and when the parser is instantiated, it must be possible to fully evaluate the expressions supplied for these parameters at compilation time.

Consider the following example:

```
parser GenericParser(packet_in b, out Packet_header p)
                      (bool udpSupport) {   // constructor parameters
    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            16w0x0800: ipv4;
        }
    }
    state ipv4 {
        b.extract(p.ipv4);
        transition select(p.ipv4.protocol) {
```

```
            6: tcp;
            17: tryudp;
        }
    }
    state tryudp {
        transition select(udpSupport) {
            false: accept;
            true : udp;
        }
    }
    state udp {
         ...
    }
}
```

When instantiating the `GenericParser` it is necessary to supply a value for the `udpSupport` parameter, as in the following example:

```
// topParser is a GenericParser where udpSupport = false
GenericParser(false) topParser;
```

13.1. Direct type invocation

Controls and parsers are often instantiated exactly once. As a light syntactic sugar, control and parser declarations with no constructor parameters may be applied directly, as if they were an instance. This has the effect of creating and applying a local instance of that type.

```
control Callee( ... ) { ... }

control Caller( ... )( ... ) {
    apply {
        Callee.apply( ... ); // callee is treated as an instance
    }
}
```

The definition of `Caller` is equivalent to the following.

```
control Caller( ... )( ... ) {
    @name("Callee") Callee() Callee_inst; // local instance of Callee
    apply {
        Callee_inst.apply( ... );          // Callee_inst is applied
    }
}
```

This feature is intended to streamline the common case where a type is instantiated exactly once. For completeness, the behavior of directly invoking the same type more than once is defined as follows.

- Direct type invocation in different scopes will result in different local instances with different fully-qualified control names.
- In the same scope, direct type invocation will result in a different local instance per invocation—however, instances of the same type will share the same global name, via the `@name` annotation. If the type contains controllable entities, then invoking it directly more than once in the same scope is illegal, because it will produce multiple controllable entities with the same fully-qualified control name.

See Section 16.3.2 for details of `@name` annotations.

14. Deparsing

The inverse of parsing is deparsing, or packet construction. P4 does not provide a separate language for packet deparsing; deparsing is done in a `control` block that has at least one parameter of type `packet_out`.

For example, the following code sequence writes first an Ethernet header and then an IPv4 header into a `packet_out`:

```
control TopDeparser(inout Parsed_packet p, packet_out b) {
    apply {
        b.emit(p.ethernet);
        b.emit(p.ip);
    }
}
```

Emitting a header appends the header to the `packet_out` only if the header is valid. Emitting a header stack will emit all elements of the stack in order of increasing indexes.

14.1. Data insertion into packets

The `packet_out` datatype is defined in the P4 core library, and reproduced below. It provides a method for appending data to an output packet called `emit`:

```
extern packet_out {
    void emit<T>(in T data);
}
```

The `emit` method supports appending the data contained in a header, header stack, `struct`, or header union to the output packet.

- When applied to a header, `emit` appends the data in the header to the packet if it is valid and otherwise behaves like a no-op.
- When applied to a header stack, `emit` recursively invokes itself to each element of the stack.
- When applied to a `struct` or header union, `emit` recursively invokes itself to each field.

It is illegal to invoke `emit` on an expression of whose type is a base type, `enum`, or `error`.

We can define the meaning of the `emit` method in pseudo-code as follows:

```
packet_out {
     byte[] data;
     unsigned lengthInBits;
     void initializeForWriting() {
          this.data.clear();
          this.lengthInBits = 0;
     }
     /// Append data to the packet. Type T must be a header, header
     /// stack, header union, or struct formed recursively from those types
     void emit<T>(T data) {
          if (isHeader(T))
               if(data.valid$) {
                    this.data.append(data);
                    this.lengthInBits += data.lengthInBits;
               }
          else if (isHeaderStack(T))
               for (e : data)
                     emit(e);
          else if (isHeaderUnion(T) || isStruct(T))
               for (f : data.fields$)
                     emit(e.f)
          // Other cases for T are illegal
     }
```

Here we use the special `valid$` identifier to indicate the hidden valid bit of headers and `fields$` to indicate the list of fields for a struct or header union. We also use standard `for` notation to iterate through the elements of a stack `(e : data)` and list of fields for header unions and structs `(f : data.fields$)`. The iteration order for a struct is the order those fields appear in the type declaration.

15. Architecture description

The architecture description must be provided by the target manufacturer in the form of a library P4 source file that contains at least one declaration for a `package`; this `package` must be instantiated by the user to construct a program for a target. For an example see the Very Simple Switch declaration from Section 5.1.

The architecture description file may pre-define data types, constants, helper package implementations, and errors. It must also declare the types of all the programmable blocks that will appear in the final target: `parser`s and `control` blocks. The programmable blocks may optionally be grouped together in packages, which can be nested.

Since some of the target components may manipulate user-defined types, which are unknown at the target declaration time, these are described using type variables, which must be used parametrically in the program—i.e., type variables are checked similar to Java generics, not C++ templates.

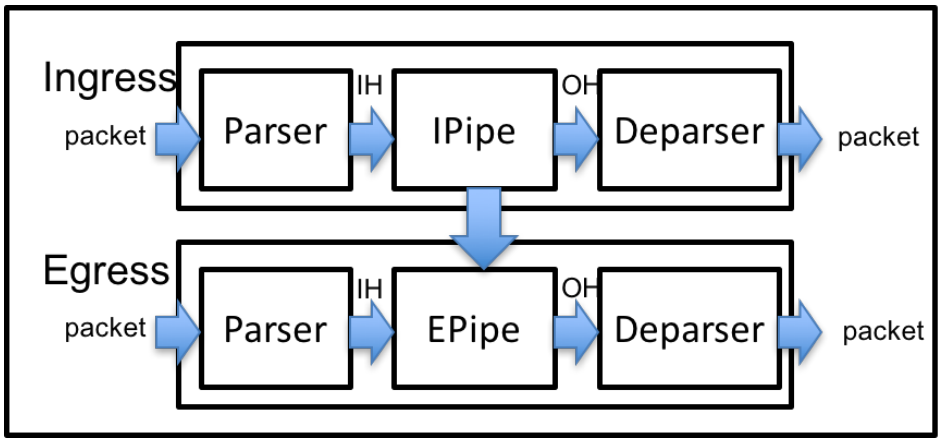

Figure 12. Fragment of example switch architecture.

15.1. Example architecture description

The following example describes a switch by using two packages, each containing a parser, a match-action pipeline, and a deparser:

```
parser Parser<IH>(packet_in b, out IH parsedHeaders);
// ingress match-action pipeline
control IPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         out OH outputHeaders,
                         out T toEgress,
                         out OutControl outCtrl);
// egress match-action pipeline
control EPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         in T fromIngress,
                         out OH outputHeaders,
                         out OutControl outCtrl);
control Deparser<OH>(in OH outputHeaders, packet_out b);
package Ingress<T, IH, OH>(Parser<IH> p,
                           IPipe<T, IH, OH> map,
                           Deparser<OH> d);
package Egress<T, IH, OH>(Parser<IH> p,
                          EPipe<T, IH, OH> map,
                          Deparser<OH> d);
package Switch<T>(Ingress<T, _, _> ingress, Egress<T, _, _> egress);
```

Just from these declarations, even without reading a precise description of the target, the programmer can infer some useful information about the architecture of the described switch, as shown in Figure 12:

- The switch contains two separate packages `Ingress` and `Egress`.
- The `Parser`, `IPipe`, and `Deparser` in the `Ingress` package are chained together in order. In addition,

the `Ingress.IPipe` block has an input of type `Ingress.IH`, which is an output of the `Ingress.Parser`.
- Similarly, the `Parser`, `EPipe`, and `Deparser` are chained in the `Egress` package.
- The `Ingress.IPipe` is connected to the `Egress.EPipe`, because the first outputs a value of type `T`, which is an input to the second. Note that the the occurrences of the type variable `T` are instantiated with the same type in `Switch`. In contrast, the `Ingress` type `IH` and the `Egress` type `IH` may be different. To force them to be the same, we could instead declare `IH` and `OH` at the switch level: `package Switch<T,IH,OH>(Ingress<T, IH, OH> ingress, Egress<T, IH, OH> egress).`

Hence, this architecture models a target switch that contains two separate channels between the ingress and egress pipeline:

- A channel that can pass data directly via its argument of type `T`. On a software target with shared memory between ingress and egress this could be implemented by passing directly a pointer; on an architecture without shared memory presumably the compiler will need to synthesize automatically serialization code.
- A channel that can pass data indirectly using a parser and deparser that serializes data into a packet and back.

15.2. Example architecture program

To construct a program for the architecture, the P4 program must instantiate a top-level `package` by passing values for all its arguments creating a variable called `main` in the top-level namespace. The types of the arguments must match the types of the parameters—after a suitable substitution of the type variables. The type substitution can be expressed directly, using type specialization, or can be inferred by a compiler, using a unification algorithm like Hindley-Milner.

For example, given the following type declarations:

```
parser Prs<T>(packet_in b, out T result);
control Pipe<T>(in T data);
package Switch<T>(Prs<T> p, Pipe<T> map);
```

and the following declarations:

```
parser P(packet_in b, out bit<32> index) { ... }
control Pipe1(in bit<32> data) { ... }
control Pipe2(in bit<8> data) { ... }
```

The following is a legal declaration for the top-level target:

```
Switch(P(), Pipe1()) main;
```

And the following is illegal:

```
Switch(P(), Pipe2()) main;
```

The latter declaration is incorrect because the parser `P` requires `T` to be `bit<32>`, while `Pipe2` requires `T` to be `bit<8>`.

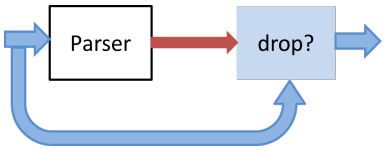

Figure 13. A packet filter target model. The parser computes a Boolean value, which is used to decide whether the packet is dropped.

The user can also explicitly specify values for the type variables (otherwise the compiler has to infer values for these type variables):

```
Switch<bit<32>>(P(), Pipe1()) main;
```

15.3. A Packet Filter Model

To illustrate the versatility of P4 architecture description language, we give an example of another architecture, which models a packet filter that makes a drop/no drop decision based only on the computation in a P4 parser, as shown in Figure 13.

This model could be used to program packet filters running in the Linux kernel. For example, we could replace the TCP dump language with the much more powerful P4 language; P4 can seamlessly support new protocols, while providing complete "type safety" during packet processing. For such a target the P4 compiler could generate an eBPF (Extended Berkeley Packet Filter) program, which is injected by the TCP dump utility into the Linux kernel, and executed by the EBPF kernel JIT compiler/runtime.

In this case the target is the Linux kernel, and the architecture model is a packet filter.

The declaration for this architecture is as follows:

```
parser Parser<H>(packet_in packet, out H headers);
control Filter<H>(inout H headers, out bool accept);

package Program<H>(Parser<H> p, Filter<H> f);
```

16. P4 abstract machine: Evaluation

The evaluation of a P4 program is done in two stages:

- static evaluation: at compile time the P4 program is analyzed and all stateful blocks are instantiated.
- dynamic evaluation: at runtime each P4 functional block is executed atomically, in isolation, when it receives control from the architecture

16.1. Compile-time known values

The following are compile-time known values:

- Integer literals, Boolean literals, and string literals.
- Identifiers declared in an `error`, `enum`, or `match_kind` declaration.
- The `default` identifier.
- The `size` field of a value with type header stack.
- The _ identifier when used as a `select` expression label
- Identifiers that represent declared types, actions, tables, parsers, controls, or packages.
- List expression where all components are compile-time known values.
- Instances constructed by instance declarations (Section 9.3) and constructor invocations.
- The following expressions (`+`, `-`, `*`, `/`, `%`, `cast`, `!`, `&`, `|`, `&&`, `||`, `<<`, `>>`, `~`, `>`, `<`, `==`, `!=`, `<=`, `>=`, `++`, `[:]`) when their operands are all compile-time known values.
- Identifiers declared as constants using the `const` keyword.

16.2. Compile-time Evaluation

Evaluation of a program proceeds in order of declarations, starting in the top-level namespace:

- All declarations (e.g., `parsers`, `controls`, types, constants) evaluate to themselves.
- Each `table` evaluates to a table instance.
- Constructor invocations evaluate to stateful objects of the corresponding type. For this purpose, all constructor arguments are evaluated recursively and bound to the constructor parameters. Constructor arguments must be compile-time known values. The order of evaluation of the constructor arguments should be unimportant — all evaluation orders should produce the same results.
- Instantiations evaluate to named stateful objects.
- The instantiation of a `parser` or `control` block recursively evaluates all stateful instantiations declared in the block.
- The result of the program's evaluation is the value of the top-level `main` variable.

Note that all stateful values are instantiated at compilation time.

As an example, consider the following program fragment:

```
// architecture declaration
parser P(...);
control C(...);
control D(...);

package Switch(P prs, C ctrl, D dep);

extern Checksum16 { ...}

// user code
Checksum16() ck16; // checksum unit instance

parser TopParser(...)(Checksum16 unit) { ...}
control Pipe(...) { ...}
control TopDeparser(...)(Checksum16 unit) { ...}
```

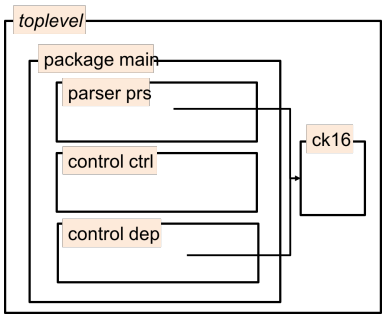

Figure 14. Evaluation result.

```
Switch(TopParser(ck16),
        Pipe(),
        TopDeparser(ck16)) main;
```

The evaluation of this program proceeds as follows:

1. The declarations of `P`, `C`, `D`, `Switch`, and `Checksum16` all evaluate to themselves.
2. The `Checksum16() ck16` instantiation is evaluated and it produces an object named `ck16` with type `Checksum16`.
3. The declarations for `TopParser`, `Pipe`, and `TopDeparser` evaluate as themselves.
4. The `main` variable instantiation is evaluated:
   (a) The arguments to the constructor are evaluated recursively
   (b) `TopParser(ck16)` is a constructor invocation
   (c) Its argument is evaluated recursively; it evaluates to the `ck16` object
   (d) The constructor itself is evaluated, leading to the instantiation of an object of type `TopParser`
   (e) Similarly, `Pipe()` and `TopDeparser(ck16)` are evaluated as constructor calls.
   (f) All the arguments of the `Switch` package constructor have been evaluated (they are an instance of `TopParser`, an instance of `Pipe`, and an instance of `TopDeparser`). Their signatures are matched with the `Switch` declaration.
   (g) Finally, the `Switch` constructor can be evaluated. The result is an instance of the `Switch` package (that contains a `TopParser` named `prs` the first parameter of the `Switch`; a `Pipe` named `ctrl`; and a `TopDeparser` named `dep`).
5. The result of the program evaluation is the value of the `main` variable, which is the above instance of the `Switch` `package`.

Figure 14 shows the result of the evaluation in a graphical form. The result is always a graph of instances. There is only one instance of `Checksum16`, called `ck16`, shared between the `TopParser` and `TopDeparser`. Whether this is possible is architecture-dependent. Specific target compilers may require distinct checksum units to be used in distinct blocks.

16.3. Control plane names

Every controllable entity exposed in a P4 program must be assigned a unique, fully-qualified name, which the control plane may use to interact with that entity. The following entities are controllable.

- tables
- keys
- actions
- extern instances

A fully qualified name consists of the local name of a controllable entity prepended with the fully qualified name of its enclosing namespace. Hence, the following program constructs, which enclose controllable entites, must themselves have unique, fully-qualified names.

- control instances
- parser instances

Evaluation may create multiple instances from one type, each of which must have a unique, fully-qualified name.

16.3.1. Computing control names

The fully-qualified name of a construct is derived by concatenating the fully-qualified name of its enclosing construct with its local name. Constructs with no enclosing namespace, i.e. those defined at the global scope, have the same local and fully-qualified names. The local names of controllable entities and enclosing constructs are derived from the syntax of a P4 program as follows.

16.3.1.1. Tables For each `table` construct, its syntactic name becomes the local name of the table. For example:

```
control c(...)() {
     table t { ... }
}
```

This table's local name is `t`.

16.3.1.2. Keys Syntactically, table keys are expressions. For simple expressions, the local key name can be generated from the expression itself. In the following example, the table `t` has keys with names `data.f1` and `hdrs[3].f2`.

```
table t {
     keys = {
          data.f1 : exact;
          hdrs[3].f2 : exact;
     }
     actions = { ... }
}
```

The following kinds of expressions have local names derived from their syntactic names:

Kind	Example	Name
The `isValid()` method.	`h.isValid()`	`"h.isValid()"`
Array accesses.	`header_stack[1]`	`"header_stack[1]"`
Constants.	`1`	`"1"`
Field projections.	`data.f1`	`"data.f1"`
Slices.	`f1[3:0]`	`"f1[3:0]"`

All other kinds of expressions must be annotated with a `@name` annotation (Section 17.1.2), as in the following example.

```
table t {
     keys = {
          data.f1 + 1 : exact @name("f1_mask");
     }
     actions = { ... }
}
```

Here, the `@name("f1_mask")` annotation assigns the local name `"f1_mask"` to this key.

16.3.1.3. Actions For each `action` construct, its syntactic name is the local name of the action. For example:

```
control c(...)() {
     action a(...) { ... }
}
```

This action's local name is `a`.

16.3.1.4. Instances The local names of `extern`, `parser`, and `control` instances are derived based on how the instance is used. If the instance is bound to a name, that name becomes its local control plane name. For example, if `control C` is declared as,

```
control C(...)() { ... }
```

and instantiated as,

```
C() c_inst;
```

then the local name of the instance is `c_inst`.

Alternatively, if the instance is created as an actual argument, then its local name is the name of the formal parameter to which it will be bound. For example, if `extern E` and `control C` are declared as,

```
extern E { ... }
control C( ... )(E e_in) { ... }
```

and instantiated as,

```
C(E()) c_inst;
```

then the local name of the extern instance is `e_in`.

If the construct being instantiated is passed as an argument to a package, the instance name is derived from the user-supplied type definition when possible. In the following example, the local name of the instance of `MyC` is `c`, and the local name of the `extern` is `e2`, not `e1`.

```
extern E { ... }
control ArchC(E e1);
package Arch(ArchC c);

control MyC(E e2)() { ... }
Arch(MyC()) main;
```

Note that in this example, the architecture will supply an instance of the extern when it applies the instance of `MyC` passed to the `Arch` package. The fully-qualified name of that instance is `main.c.e2`.

Next, consider a larger example that demonstrates name generation when there are multiple instances.

```
control Callee() {
     table t { ... }
     apply { t.apply(); }
}
control Caller() {
     Callee() c1;
     Callee() c2;
     apply {
         c1.apply();
         c2.apply();
     }
}
control Simple();
package Top(Simple s);
Top(Caller()) main;
```

The compile-time evaluation of this program produces the structure in Figure 15. Notice that there are two instances of the `table t`. These instances must both be exposed to the control plane. To name an object in this hierarchy, one uses a path composed of the names of containing instances. In this case, the two tables have names `s.c1.t` and `s.c2.t`, where `s` is the name of the argument to the package instantiation, which is derived from the name of its corresponding formal parameter.

16.3.2. Annotations controlling naming

Control plane-related annotations (Section 17.1.2) can alter the names exposed to the control plane in the following ways.

- The `@hidden` annotation hides a controllable entity from the control plane. This is the only case in which a controllable entity is not required to have a unique, fully-qualified name.

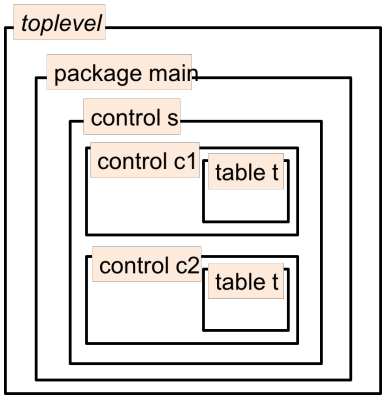

Figure 15. Evaluating a program that has several instantiations of the same component.

- The `@name` annotation may be used to change the local name of a controllable entity.
- The `@globalname` annotation my be used to change the global name of a controllable entity.

Programs that yield the same fully-qualified name for two different controllable entities are invalid. Care must be taken with `@globalname` in particular: If a type contains a `@globalname` annotation and is instantiated twice, the two instances will have the same fully-qualified name.

16.3.3. Recommendations

The control plane may refer to a controllable entity by a postfix of its fully qualified name when it is unambiguous in the context in which it is used. Consider the following example.

```
control c( ... )() {
     action a ( ... ) { ... }
     table t {
          keys = { ... }
          actions = { a; } }
}
c() c_inst;
```

Control plane software may refer to action `c_inst.a` as `a` when inserting rules into table `c_inst.t`, because it is clear from the definition of the table which action `a` refers to.

Not all unambiguous postfix shortcuts are recommended. For instance, consider the first example in Section 16.3. One might be tempted to refer to `s.c1` simply as `c1`, as no other instance named `c1` appears in the program. However, this leads to a brittle program since future modifications can never introduce an instance named `c1`, or include libraries of P4 code that contain instances with that name.

16.4. Dynamic evaluation

The dynamic evaluation of a P4 program is orchestrated by the architecture model. Each architecture model needs to specify the order and the conditions under which the various P4 component programs

are dynamically executed. For example, in the Simple Switch example from Section 5.1 the execution flow goes `Parser->Pipe->Deparser`.

Once a P4 execution block is invoked its execution proceeds until termination according to the semantics defined in this document.

16.4.1. Concurrency model

A typical packet processing system needs to execute multiple simultaneous logical "threads." At the very least there is a thread executing the control plane, which can modify the contents of the tables. Architecture specifications should describe in detail the interactions between the control-plane and the data-plane. The data plane can exchange information with the control plane through `extern` function and method calls. Moreover, high-throughput packet-processing systems may be processing multiple packets simultaneously, e.g., in a pipelined fashion, or concurrently parsing a first packet while performing match-action operations on a second packet. This section specifies the semantics of P4 programs with respect to such concurrent executions.

Each top-level `parser` or `control` block is executed as a separate thread when invoked by the architecture. All the parameters of the block and all local variables are thread-local—i.e., each thread has a private copy of these resources. This applies to the `packet_in` and `packet_out` parameters of parsers and deparsers.

As long as a P4 block uses only thread-local storage (e.g., metadata, packet headers, local variables), its behavior in the presence of concurrency is identical with the behavior in isolation, since any interleaving of statements from different threads must produce the same output.

In contrast, `extern` blocks instantiated by a P4 program are global, shared across all threads. If `extern` blocks mediate access to state (e.g., counters, registers)—i.e., the methods of the `extern` block read and write state, these stateful operations are subject to data races. P4 mandates the following behaviors:

- Execution of an action is atomic—i.e., the other threads can "see" the state as it is either before the start of the action or after the completion of the action.
- Execution of a method call on an extern instance is atomic.

To allow users to express atomic execution of larger code blocks, P4 provides an `@atomic` annotation, which can be applied to block statements, parser states, control blocks, or whole parsers.

Consider the following example:

```
extern Register { ... }
control Ingress() {
  Register() r;
  table flowlet { /* read state of r in an action */ }
  table new_flowlet { /* write state of r in an action */ }
  apply {
    @atomic {
        flowlet.apply();
        if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TIMEOUT)
            new_flowlet.apply();
}}}
```

This program accesses an extern object `r` of type `Register` in actions invoked from tables `flowlet` (reading) and `new_flowlet` (writing). Without the `@atomic` annotation these two operations would not execute atomically: a second packet may read the state of `r` before the first packet had a chance to update it.

A compiler backend must reject a program containing `@atomic` blocks if it cannot implement the atomic execution of the instruction sequence. In such cases, the compiler should provide reasonable diagnostics.

17. Annotations

Annotations are similar to C# attributes and Java annotations. They are a simple mechanism for extending the P4 language to some limited degree without changing the grammar. To some degree they subsume the C `#pragmas`. Annotations can be added to types, fields, variables, etc. using the `@` syntax (as shown explicitly in the P4 grammar):

```
optAnnotations
     : /* empty */
     | annotations
     ;

annotations
     : annotation
     | annotations annotation
     ;

annotation
     : '@' name
     | '@' name '(' expressionList ')'
     ;
```

17.1. Predefined annotations

Annotation names that start with lowercase letters are reserved for the standard library and architecture. This document pre-defines a set of "standard" annotations. We expect that this list will grow. We encourage custom architectures to define annotations starting with a manufacturer prefix: e.g., an organization named X would use annotations named like `@X_annotation`

17.1.1. Annotations on the table action list

The following two annotations can be used to give additional information to the compiler and control-plane about actions in a table. They have no arguments.

- `@tableonly`: actions with this annotation can only appear within the table, and never as default action.
- `@defaultonly`: actions with this annotation can only appear in the default action, and never in the table.

```
table t {
     actions = {
         a,                 // can appear anywhere
         @tableonly b,    // can only appear in the table
         @defaultonly c, // can only appear in the default action
     }
     ...
}
```

17.1.2. Control-plane API annotations

The `@name` annotation directs the compiler to use a different local name when generating the external APIs used to manipulate a language element from the control plane. It must have a string literal argument. In the following example, the fully-qualified name of the table is `c_inst.t1`.

```
control c( ... )() {
     @name("t1") table t { ... }
     apply { ... }
}
c() c_inst;
```

The `@globalname` annotation acts like the `@name` annotation, except it overrides the fully-qualified name (not just the local name) for the annotated element. In the following example, the fully-qualified name of the table is `foo.bar`.

```
control c( ... )() {
     @globalname("foo.bar") table t { ... }
     apply { ... }
}
c() c_inst;
```

The `@hidden` annotation hides a controllable entity, e.g. a table, key, action, or extern, from the control plane. This effectively removes its fully-qualified name (Section 16.3). It does not take any arguments.

17.1.2.1. Restrictions Each element may be annotated with at most one `@name`, `@globalname`, or `@hidden` annotation, and each control plane name must refer to at most one controllable entity. This is of special concern when using the `@globalname` annotation: If a type containing a `@globalname` annotation is instantiated more than once, it will result in the same global name referring to two controllable entities.

```
control noargs();
package top(noargs c1, noargs c2);

control c() {
     @globalname("foo.bar") table t { ... }
```

```
    apply { ... }
}
top(c(), c()) main;
```

Without the `@globalname` annotation, this program would produce two controllable entities with fully-qualified names `main.c1.t` and `main.c2.t`. However, the `@globalname("foo.bar")` annotation renames table `t` in both instances to `foo.bar`, resulting in one name that refers to two controllable entities, which is illegal.

17.1.3. Concurrency control annotations

The `@atomic` annotation, described in Section 16.4.1 can be used to enforce the atomic execution of a code block.

17.2. Target-specific annotations

Each P4 compiler implementation can define additional annotations specific to the target of the compiler. The syntax of the annotations should conform to the above description. The semantics of such annotations is target-specific. They could be used in a similar way to pragmas in other languages.

The P4 compiler should provide:

- Errors when annotations are used incorrectly (e.g., an annotation expecting a parameter but used without arguments, or with arguments of the wrong type
- Warnings for unknown annotations.

A. Appendix: P4 reserved keywords

The following table shows all P4 reserved keywords. Some identifiers are treated as keywords only in specific contexts (e.g., the keyword `actions`).

`action`	`apply`	`bit`	`bool`
`const`	`control`	`default`	`else`
`enum`	`error`	`extern`	`exit`
`false`	`header`	`header_union`	`if`
`in`	`inout`	`int`	`match_kind`
`package`	`parser`	`out`	`return`
`select`	`state`	`struct`	`switch`
`table`	`transition`	`true`	`tuple`
`typedef`	`varbit`	`verify`	`void`

B. Appendix: P4 core library

The P4 core library contains declarations that are useful to most programs.

For example, the core library includes the declarations of the predefined `packet_in` and `packet_out` extern objects, used in parsers and deparsers to access packet data.

```
/// Standard error codes.  New error codes can be declared by users.
error {
    NoError,            /// No error.
    PacketTooShort,     /// Not enough bits in packet for 'extract'.
    NoMatch,            /// 'select' expression has no matches.
    StackOutOfBounds,   /// Reference to invalid element of a header stack.
    HeaderTooShort,     /// Extracting too many bits into a varbit field.
    ParserTimeout       /// Parser execution time limit exceeded.
}
extern packet_in {
    /// Read a header from the packet into a fixed-sized header @hdr
    /// and advance the cursor.
    /// May trigger error PacketTooShort or StackOutOfBounds.
    /// @T must be a fixed-size header type
    void extract<T>(out T hdr);
    /// Read bits from the packet into a variable-sized header @variableSizeHeader
    /// and advance the cursor.
    /// @T must be a header containing exactly 1 varbit field.
    /// May trigger errors PacketTooShort, StackOutOfBounds, or HeaderTooShort.
    void extract<T>(out T variableSizeHeader,
                    in bit<32> variableFieldSizeInBits);
    /// Read bits from the packet without advancing the cursor.
    /// @returns: the bits read from the packet.
    /// T may be an arbitrary fixed-size type.
    T lookahead<T>();
    /// Advance the packet cursor by the specified number of bits.
    void advance(in bit<32> sizeInBits);
    /// @return packet length in bytes.  This method may be unavailable on
    /// some target architectures.
    bit<32> length();
}
extern packet_out {
    /// Write @data into the output packet, skipping invalid headers
    /// and advancing the cursor
    /// @T can be a header type, a header stack, a header_union, or a struct
    /// containing fields with such types.
    void emit<T>(in T data);
}
action NoAction() {}
/// Standard match kinds for table key fields.
/// Some architectures may not support all these match kinds.
/// Architectures can declare additional match kinds.
match_kind {
    /// Match bits exactly.
    exact,
```

```
    /// Ternary match, using a mask.
    ternary,
    /// Longest-prefix match.
    lpm
}
```

C. Appendix: Checksums

There are no built-in constructs in $P4_{16}$ for manipulating packet checksums. We expect that checksum operations can be expressed as `extern` library objects that are provided in target-specific libraries. The standard architecture library should provide such checksum units.

For example, one could provide an incremental checksum unit `Checksum16` (also described in the VSS example in Section 5.2.4) for computing 16-bit one's complement using an `extern` object with a signature such as:

```
extern Checksum16 {
    Checksum16();                // constructor
    void clear();                // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

IP checksum verification could be done in a parser as:

```
ck16.clear();             // prepare checksum unit
ck16.update(h.ipv4);     // write header
verify(ck16.get() == 16w0, error.IPv4ChecksumError); // check for 0 checksum
```

IP checksum generation could be done as:

```
h.ipv4.hdrChecksum = 16w0;
ck16.clear();
ck16.update(h.ipv4);
h.ipv4.hdrChecksum = ck16.get();
```

Moreover, some switch architectures do not perform checksum verification, but only update checksums incrementally to reflect packet modifications. This could be achieved as well, as the following P4 program fragments illustrates:

```
ck16.clear();
ck16.update(h.ipv4.hdrChecksum);  // original checksum
ck16.remove( { h.ipv4.ttl, h.ipv4.proto } );
h.ipv4.ttl = h.ipv4.ttl - 1;
```

```
ck16.update( { h.ipv4.ttl, h.ipv4.proto } );
h.ipv4.hdrChecksum = ck16.get();
```

D. Appendix: Restrictions on compile time and run time calls

This appendix summarizes restrictions on compile time and run time calls that can be made. Many of them are described earlier in this document, but are collected here for easy reference.

The stateful types of objects in P4$_{16}$ are packages, parsers, controls, externs, and tables. All other types are referred to as "value types" here.

Some guiding principles:

- Controls are not allowed to call parsers, and vice versa, so there is no use in passing one type to the other in constructor parameters or run-time parameters.
- At run time, after a control is called, and before that call is complete, there can be no recursive calls between controls, nor from a control to itself. Similarly for parsers. There can be loops among states within a single parser.
- Externs are not allowed to call parsers or controls, so there is no use in passing objects of those types to them.
- Tables are always instantiated directly in their enclosing control, and cannot be instantiated at the top level. There is no syntax for specifying parameters that are tables. Tables are only intended to be used from within the control where they are defined.

A note on recursion: It is expected that some architectures will define capabilities for recirculating a packet to be processed again as if it were a newly arriving packet, or to make "clones" of packets that are then processed by parsers and/or control blocks that the original packet has already completed. This does not change the notes above on recursion that apply while a parser or control is executing.

The first table lists restrictions on what types can be passed as constructor parameters to other types.

	can be a constructor parameter for this type			
This type	package	parser	control	extern
package	yes	no	no	no
parser	yes	yes	no	no
control	yes	no	yes	no
extern	yes	yes	yes	yes
table	no	no	no	no
value types	yes	yes	yes	yes

The next table lists restrictions on where one may perform instantiations (see Section 9.3) of different types. The answer for `package` is always "no" because there is no "inside a package" where instantiations can be written in P4$_{16}$. One can definitely make constructor calls and use instances of stateful types as parameters when instantiating a package, and restrictions on those types are in the table above.

For externs, one can only specify their interface in P4$_{16}$, not their implementation. Thus there is no place to instantiate objects within an extern.

You may declare variables and constants of any of the value types within a parser or control (see Section 9.2 for more details). Declaring a variable or constant is not the same as instantiation, hence

the answer "N/A" (for not applicable) in those table entries. Variables may not be declared at the top level of your program, but constants may.

	can be instantiated in this place				
This type	top level	package	parser	control	extern
package	yes	no	no	no	no
parser	no	no	yes	no	no
control	no	no	no	yes	no
extern	yes	no	yes	yes	no
table	no	no	no	yes	no
value types	N/A	N/A	N/A	N/A	N/A

The next table lists restrictions on what types can be passed as run-time parameters to other callable things that have run-time parameters: parsers, controls, extern methods, and actions.

	can be a run-time parameter to this callable thing			
This type	parser	control	method	action
package	no	no	no	no
parser	no	no	no	no
control	no	no	no	no
extern	yes	yes	yes	no
table	no	no	no	no
value types	yes	yes	yes	yes

Extern method calls may return only return a value that is a value type, or no value at all (specified by a return type of `void`).

The next table lists restrictions on what kinds of calls can be made from which places in a P4 program. Calling a parser, control, or table means invoking its `apply()` method. The row for `extern` describes where extern method calls can be made from.

One way that an extern can be called from the top level of a parser or control is in an initializer expression for a declared variable, e.g. `bit<32> x = rand.get();`.

	can be called at run time from this place in a P4 program				
This type	parser state	control apply block	parser or control top level	action	extern
package	N/A	N/A	N/A	N/A	N/A
parser	yes	no	no	no	no
control	no	yes	no	no	no
extern	yes	yes	yes	yes	no
table	no	yes	no	no	no
action	no	yes	no	yes	no
value types	N/A	N/A	N/A	N/A	N/A

There may not be any recursion in calls, neither by a thing calling itself directly, nor mutual recursion.

An extern can never cause any other type of P4 program object to be called. See Section 6.7.1.

Actions may be called directly from a control `apply` block.

Note that while the extern row shows that extern methods can be called from many places, particular externs may have additional restrictions not listed in this table. Any such restrictions should be documented in the description for each extern, as part of the documentation for the architecture that

defines the extern.

In many cases, the restriction will be “from a parser state only” or “from a control apply block or action only”, but it may be even more restrictive, e.g. only from a particular kind of control block instantiated in a particular role in an architecture.

E. Appendix: Open Issues

There are a number of open issues that are currently under discussion in the P4 design working group. A brief summary of these issues is highlighted in this section. We seek input on these issues from the community, and encourage experimenting with different implementations in the compiler before converging on the specification.

E.1. Portable Switch Architecture

Portability and composability are critical to P4's long-term success: - Composability: implement different features, such as In-band Network Telemetry (INT), Network Virtualization, and Load Balancing in separate P4 programs written for the PSA, should easily inter-operate when invoked from a top-level program. - Portability: a P4 implementation of a certain function, such as INT, against PSA should work consistently across architectures that support the PSA. To that end, a specification for an architecture definition that enables programmers to write composable P4 programs, called the Portable Switch Architecture, is being developed by the Architecture Working Group.

E.2. Generalized switch statement behavior

$P4_{16}$ includes both `switch` statements 10.7 and `select` expressions 11.6. There are real differences in the current version of the language – expression vs. statement, and the latter must evaluate to a state value.

We propose generalizing `switch` statements to match the design used in most programming language: a multi-way conditional that executes the first branch that matches from a list of cases.

```
switch(e1,...,en) {
  pat_1 : stmt1;
  ...
  pat_m : stmtm;
}
```

Here, the value being scrutinized is given by a tuple `(e1,...,en)`, and the patterns are given by expressions that denote sets of values. The value matches a branch if it is an element of the set denoted by the pattern. Unlike C and C++, there is no break statement so control “falls through” to the next case only when there is no statement associated with the case label.

This design is intended to capture the standard semantics of `switch` statements as well as a common idiom in P4 parsers where they are used to control transitions to different parser states depending on the values of one or more previously-parsed values. Using `switch` statements, we can also generalize the design for parsers, eliminating select and lifting most restrictions on which kinds of statements may appear in a state. In particular, we allow conditional statements and `select` statements, which may be nested arbitrarily. This language can be translated into more restricted versions, where the body of each

state comprised a sequence of variable declarations, assignments, and method invocations followed by a single`transition` statement by introducing new states.

We also generalize the design for processing of table hit/miss and actions in control blocks, by generating implicit types for actions and results.

The counter-argument to this proposal is that the semantics of `select` in the parser is sufficiently distinct from the `switch` statement, and moreover these are constructs that network programmers are already familiar with, and they are typically mapped very efficiently onto a variety of targets.

E.3. Undefined behaviors

The presence of undefined behavior has caused numerous problems in languages like C and HTML, including bugs and serious security vulnerabilities. There are a few places where evaluating a P4 program can result in undefined behaviors: out parameters, uninitialized variables, accessing header fields of invalid headers, and accessing header stacks with an out of bounds index. We think we should make every attempt to avoid undefined behaviors in $P4_{16}$, and therefore we propose to strengthen the wording in the specification, such that by default, we rule out programs that exhibit the behaviors mentioned above. Given the concern for performance, we propose to define compiler flags and/or pragmas that can override the safe behavior. However, our expectation is that programmers should be guided toward writing safe programs, and encouraged to think harder when excepting from the safe behavior.

E.4. Structured Iteration

Introducing a `foreach` style iterator for operating over header stacks will alleviate the need of using C preprocessor directives to specify the size of header stacks.

For example:

```
foreach hdr in hdrs {
  ... operations over HDR ...
}
```

Since the stacks are always known statically (at compile-time), the compiler could transform the `foreach` statement into the replicated code with explicit index references at compile-time. This has the advantage of allowing the code to be written without regard to a parameterized header stack length.

Since the compiler can statically determine the number of operations that would result from the `foreach` it can also reject a program if the result requires more action resources than are available, or can split the action code up to fit available resources as needed.

F. Appendix: P4 grammar

This is the grammar of $P4_{16}$ written using the YACC/bison language. Absent from this grammar is the precedence of various operations.

The grammar is actually ambiguous, so the lexer and the parser must collaborate for parsing the language. In particular, the lexer must be able to distinguish two kinds of identifiers:

- Type names previously introduced (`TYPE` tokens)
- Regular identifiers (`IDENTIFIER` token)

The parser has to use a symbol table to indicate to the lexer how to parse subsequent appearances of identifiers. For example, given the following program fragment:

```
typedef bit<4> t;
struct s { ...}
t x;
parser p(bit<8> b) { ... }
```

The lexer has to return the following terminal kinds:

```
t - TYPE
s - TYPE
x - IDENTIFIER
p - TYPE
b - IDENTIFIER
```

This grammar has been heavily influenced by limitations of the Bison parser generator tool.

Several other constant terminals appear in these rules:

```
- SHL is <<
- LE is <=
- GE is >=
- NE is !=
- EQ is ==
- PP is ++
- AND is &&
- OR is ||
- MASK is &&&
- RANGE is ..
- DONTCARE is _
```

The STRING_LITERAL token corresponds to a string literal enclosed within double quotes, as described in Section 6.3.3.3.

All other terminals are uppercase spellings of the corresponding keywords. For example, RETURN is the terminal returned by the lexer when parsing the keyword return.

```
p4program
     : /* empty */
     | p4program declaration
     | p4program ';'  /* empty declaration */
     ;

declaration
     : constantDeclaration
     | externDeclaration
     | actionDeclaration
     | parserDeclaration
```

```
    | typeDeclaration
    | controlDeclaration
    | instantiation
    | errorDeclaration
    | matchKindDeclaration
    ;

nonTypeName
    : IDENTIFIER
    | APPLY
    | KEY
    | ACTIONS
    | STATE
    ;

name
    : nonTypeName
    | TYPE
    ;

optAnnotations
    : /* empty */
    | annotations
    ;

annotations
    : annotation
    | annotations annotation
    ;

annotation
    : '@' name
    | '@' name '(' expressionList ')'
    ;

parameterList
    : /* empty */
    | nonEmptyParameterList
    ;

nonEmptyParameterList
    : parameter
    | nonEmptyParameterList ',' parameter
    ;

parameter
```

```
    : optAnnotations direction typeRef name
    ;

direction
    : IN
    | OUT
    | INOUT
    | /* empty */
    ;

packageTypeDeclaration
    : optAnnotations PACKAGE name optTypeParameters
      '(' parameterList ')'
    ;

instantiation
    : typeRef '(' argumentList ')' name ';'
    | annotations typeRef '(' argumentList ')' name ';'
    ;

optConstructorParameters
    : /* empty */
    | '(' parameterList ')'
    ;

dotPrefix
    : '.'
    ;

/*************************** PARSER *****************************/

parserDeclaration
    : parserTypeDeclaration optConstructorParameters
      /* no type parameters allowed in the parserTypeDeclaration */
      '{' parserLocalElements parserStates '}'
    ;

parserLocalElements
    : /* empty */
    | parserLocalElements parserLocalElement
    ;

parserLocalElement
    : constantDeclaration
    | variableDeclaration
    | instantiation
```

```
    ;

parserTypeDeclaration
    : optAnnotations PARSER name optTypeParameters '(' parameterList ')'
    ;

parserStates
    : parserState
    | parserStates parserState
    ;

parserState
    : optAnnotations STATE name '{' parserStatements transitionStatement '}'
    ;

parserStatements
    : /* empty */
    | parserStatements parserStatement
    ;

parserStatement
    : assignmentOrMethodCallStatement
    | directApplication
    | parserBlockStatement
    | constantDeclaration
    | variableDeclaration
    ;

parserBlockStatement
    : optAnnotations '{' parserStatements '}'
    ;

transitionStatement
    : /* empty */
    | TRANSITION stateExpression
    ;

stateExpression
    : name ';'
    | selectExpression
    ;

selectExpression
    : SELECT '(' expressionList ')' '{' selectCaseList '}'
    ;
```

```
selectCaseList
     : /* empty */
     | selectCaseList selectCase
     ;

selectCase
     : keysetExpression ':' name ';'
     ;

keysetExpression
     : tupleKeysetExpression
     | simpleKeysetExpression
     ;

tupleKeysetExpression
     : '(' simpleKeysetExpression ',' simpleExpressionList ')'
     ;

simpleExpressionList
     : simpleKeysetExpression
     | simpleExpressionList ',' simpleKeysetExpression
     ;

simpleKeysetExpression
     : expression
     | DEFAULT
     | DONTCARE
     | expression MASK expression
     | expression RANGE expression
     ;

/************************** CONTROL ************************/

controlDeclaration
     : controlTypeDeclaration optConstructorParameters
       /* no type parameters allowed in controlTypeDeclaration */
       '{' controlLocalDeclarations APPLY controlBody '}'
     ;

controlTypeDeclaration
     : optAnnotations CONTROL name optTypeParameters
       '(' parameterList ')'
     ;

controlLocalDeclarations
     : /* empty */
```

```
    | controlLocalDeclarations controlLocalDeclaration
    ;

controlLocalDeclaration
    : constantDeclaration
    | actionDeclaration
    | tableDeclaration
    | instantiation
    | variableDeclaration
    ;

controlBody
    : blockStatement
    ;

/************************** EXTERN ************************/

externDeclaration
    : optAnnotations EXTERN nonTypeName optTypeParameters '{' methodPrototypes '}'
    | optAnnotations EXTERN functionPrototype ';'
    ;

methodPrototypes
    : /* empty */
    | methodPrototypes methodPrototype
    ;

functionPrototype
    : typeOrVoid name optTypeParameters '(' parameterList ')'
    ;

methodPrototype
    : optAnnotations functionPrototype ';'
    | optAnnotations TYPE '(' parameterList ')' ';'
    ;

/************************** TYPES *************************/

typeRef
    : baseType
    | typeName
    | specializedType
    | headerStackType
    | tupleType
    ;
```

```
prefixedType
    : TYPE
    | dotPrefix TYPE
    ;

typeName
    : prefixedType
    ;

tupleType
    : TUPLE '<' typeArgumentList '>'
    ;

headerStackType
    : typeName '[' expression ']'
    ;

specializedType
    : prefixedType '<' typeArgumentList '>'
    ;

baseType
    : BOOL
    | ERROR
    | BIT
    | BIT '<' INTEGER '>'
    | INT '<' INTEGER '>'
    | VARBIT '<' INTEGER '>'
    ;

typeOrVoid
    : typeRef
    | VOID
    | nonTypeName      // may be a type variable
    ;

optTypeParameters
    : /* empty */
    | '<' typeParameterList '>'
    ;

typeParameterList
    : nonTypeName
    | typeParameterList ',' nonTypeName
    ;
```

```
typeArg
     : DONTCARE
     | typeRef
     ;

typeArgumentList
     : typeArg
     | typeArgumentList ',' typeArg
     ;

typeDeclaration
     : derivedTypeDeclaration
     | typedefDeclaration
     | parserTypeDeclaration ';'
     | controlTypeDeclaration ';'
     | packageTypeDeclaration ';'
     ;

derivedTypeDeclaration
     : headerTypeDeclaration
     | headerUnionDeclaration
     | structTypeDeclaration
     | enumDeclaration
     ;

headerTypeDeclaration
     : optAnnotations HEADER name '{' structFieldList '}'
     ;

headerUnionDeclaration
     : optAnnotations HEADER_UNION name '{' structFieldList '}'
     ;

structTypeDeclaration
     : optAnnotations STRUCT name '{' structFieldList '}'
     ;

structFieldList
     : /* empty */
     | structFieldList structField
     ;

structField
     : optAnnotations typeRef name ';'
     ;
```

```
enumDeclaration
     : optAnnotations ENUM name '{' identifierList '}'
     ;

errorDeclaration
     : ERROR '{' identifierList '}'
     ;

matchKindDeclaration
     : MATCH_KIND '{' identifierList '}'
     ;

identifierList
     : name
     | identifierList ',' name
     ;

typedefDeclaration
     : optAnnotations TYPEDEF typeRef name ';'
     | optAnnotations TYPEDEF derivedTypeDeclaration name ';'
     ;

/************************** STATEMENTS ************************/

assignmentOrMethodCallStatement
     : lvalue '(' argumentList ')' ';'
     | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
     | lvalue '='  expression ';'
     ;

emptyStatement
     : ';'
     ;

returnStatement
     : RETURN ';'
     ;

exitStatement
     : EXIT ';'
     ;

conditionalStatement
     : IF '(' expression ')' statement
     | IF '(' expression ')' statement ELSE statement
     ;
```

```
// To support direct invocation of a control or parser without instantiation
directApplication
     : typeName '.' APPLY '(' argumentList ')' ';'

statement
     : assignmentOrMethodCallStatement
     | directApplication
     | conditionalStatement
     | emptyStatement
     | blockStatement
     | exitStatement
     | returnStatement
     | switchStatement
     ;

blockStatement
     : optAnnotations '{' statOrDeclList '}'
     ;

statOrDeclList
     : /* empty */
     | statOrDeclList statementOrDeclaration
     ;

switchStatement
     : SWITCH '(' expression ')' '{' switchCases '}'
     ;

switchCases
     : /* empty */
     | switchCases switchCase
     ;

switchCase
     : switchLabel ':' blockStatement
     | switchLabel ':'
     ;

switchLabel
     : name
     | DEFAULT
     ;

statementOrDeclaration
     : variableDeclaration
```

```
    | constantDeclaration
    | statement
    | instantiation
    ;

/************ TABLES *************/
tableDeclaration
    : optAnnotations TABLE name '{' tablePropertyList '}'
    ;

tablePropertyList
    : tableProperty
    | tablePropertyList tableProperty
    ;

tableProperty
    : KEY '=' '{' keyElementList '}'
    | ACTIONS '=' '{' actionList '}'
    | CONST ENTRIES '=' '{' entriesList '}' /* immutable entries */
    | optAnnotations CONST IDENTIFIER '=' initializer ';'
    | optAnnotations IDENTIFIER '=' initializer ';'
    ;

keyElementList
    : /* empty */
    | keyElementList keyElement
    ;

keyElement
    : expression ':' name optAnnotations ';'
    ;

actionList
    : /* empty */
    | actionList actionRef ';'
    ;

entriesList
    : entry
    | entriesList entry

entry
    : keysetExpression ':' actionRef optAnnotations ';'

actionRef
    : optAnnotations name
```

```
    | optAnnotations name '(' argumentList ')'
    ;

/************************ ACTION ******************************/

actionDeclaration
    : optAnnotations ACTION name '(' parameterList ')' blockStatement
    ;

/************************ VARIABLES ***************************/

variableDeclaration
    : annotations typeRef name optInitializer ';'
    | typeRef name optInitializer ';'
    ;

constantDeclaration
    : optAnnotations CONST typeRef name '=' initializer ';'
    ;

optInitializer
    : /* empty */
    | '=' initializer
    ;

initializer
    : expression
    ;

/************************ Expressions ***************************/

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList ',' argument
    ;

argument
    : expression
    | DONTCARE
    ;
```

```
expressionList
     : /* empty */
     | expression
     | expressionList ',' expression
     ;

member
     : name
     ;

prefixedNonTypeName
     : nonTypeName
     | dotPrefix nonTypeName
     ;

lvalue
     : prefixedNonTypeName
     | lvalue '.' member
     | lvalue '[' expression ']'
     | lvalue '[' expression ':' expression ']'
     ;

%left ','
%nonassoc '?'
%nonassoc ':'
%left OR
%left AND
%left '|'
%left '^'
%left '&'
%left EQ NE
%left '<' '>' LE GE
%left SHL
%left PP '+' '-'
%left '*' '/' '%'
%right PREFIX
%nonassoc ']' '(' '['
%left '.'

// Additional precedences need to be specified

expression
     : INTEGER
     | TRUE
     | FALSE
     | STRING_LITERAL
```

```
    | nonTypeName
    | '.' nonTypeName
    | expression '[' expression ']'
    | expression '[' expression ':' expression ']'
    | '{' expressionList '}'
    | '(' expression ')'
    | '!' expression
    | '~' expression
    | '-' expression
    | '+' expression
    | typeName '.' member
    | ERROR '.' member
    | expression '.' member
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | expression '+' expression
    | expression '-' expression
    | expression SHL expression         // <<
    | expression '>''>' expression      // check that >> are adjacent
    | expression LE expression          // <=
    | expression GE expression          // >=
    | expression '<' expression
    | expression '>' expression
    | expression NE expression          // !=
    | expression EQ expression          // ==
    | expression '&' expression
    | expression '^' expression
    | expression '|' expression
    | expression PP expression          // ++
    | expression AND expression         // &&
    | expression OR expression          // ||
    | expression '?' expression ':' expression
    | expression '<' typeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'
    | typeRef '(' argumentList ')'
    | '(' typeRef ')' expression
    ;
```